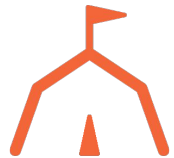
Full Stack JavaScript

JavaScript

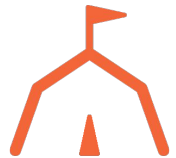
Download PDF Copy



JavaScript

- Intro to JavaScript
- Control Structures: Conditionals & Loops
- Functions
- Objects & Arrays
- Scope & Closures
- Classes

The Intro



Intro

JavaScript is a programming language developed in the late '90s.

It is a high-level, interpreted language typed dynamically.

Interpreted languages translate the code during execution.

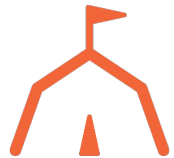
Dynamic typing allows for values reassignment.

Intro

JavaScript supports multiple styles of programming.

Imperative programming deals with statements that change the program's state (typically through conditions/loops).

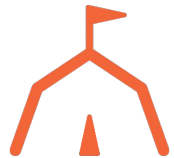
Object-oriented and procedural programming are considered mostly imperative.



Intro

Declarative programming deals with building the logic without describing the flow (generally without heavy condition/loop presence).

Functional programming is considered a declarative type.



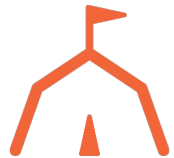
Intro

Two common approaches to creating scripts are:

- in script tags within HTML
- in files with a `.js` extension

Scripts can also be generated through tooling systems and dynamically inserted in the HTML.

Variables

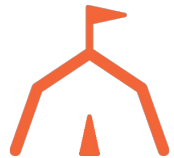


Variables

Variables are named containers for values.

There are three ways to declare a variable.

```
var name;
let favoriteFood;
const lovesJavaScript;
```



var

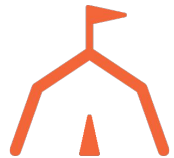
A variable declared with the `var` keyword is a standard variable.

Anything declared with `var` is function scoped. It means they are only available inside the function they're created in, or if not created inside a function, they are globally scoped.

```
var name = "Adam";
name = "Syntax the Dog";
var name = "Your Name Here";
```

Notice that the use of `var` allows for:

- reassigning
- redeclaring

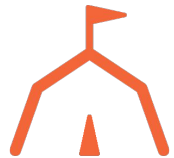


let

When declaring a variable with `let`, the following rules apply to the variable:

- the variable is scope blocked
- redeclaring the variable within the scope is not possible

```
let food = "Tacos";
food = "Nachos";
let food = "Pizza"; // SyntaxError: Identifier 'food' has already been declared
```

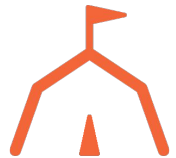


const

When declaring a variable with `const`, the following rules apply to the variable:

- all of the rules from `let` carry over to `const`
- reassigning the variable within the scope is not possible

```
const age = 32;
age = 33; // TypeError: Assignment to constant variable
const age = 34; // SyntaxError: Identifier 'age' has already been declared
```

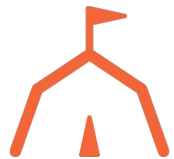


Block Scoping

Block scoping refers to variables/functions being accessible within the code **block** they are declared within.

```
{
  var name = "Adam";
  console.log(name); // "Adam"
}
console.log(name); // "Adam"

{
  let age = 32;
  console.log(age); // 32
}
console.log(age); // ReferenceError: age is not defined

{
  const favoriteFood = "nachos";
  console.log(favoriteFood); // "nachos"
}
console.log(favoriteFood); // ReferenceError: favoriteFood is not
defined
```

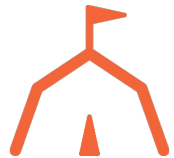
Rules, Practices, and Behaviors



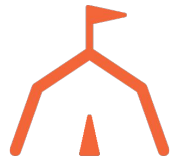
Declaring Variables

Follow these four simple rules when declaring variables.

- Must begin with a letter, _, or $
- Can contain letters, numbers, _, and $
- Names are case-sensitive
- Do not use JavaScript keywords for variable or function names



```
var currentYear = 2020; // number
let $firstName = "Grant"; // string
let _lastName = 'Chirpus'; // string - single or
double quotes for strings
const detroitIsCool = true; // boolean
DetroitIsCool
```

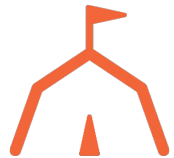


Declaring Variables

`var` is vital to understand; however, in modern times, all variables should be declared using either `let` or `const`.

Using `const` as much as possible is the preferred practice.

`let` is appropriate to use if the conditions within the program demand a value reassignment.



Hoisting

When a script file gets executed, JavaScript collects all variable and function declarations and **hoists** them to the top of whatever scope held the declaration.

It is important to remember that the value does not get brought with the declaration.

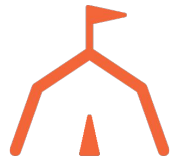
Only the declaration gets hoisted.

Consider the following code using `var`:

```
var name = "Adam";
console.log(name); // "Adam"
```

With a quick alteration, the code now acts much differently.

```
console.log(name); // undefined
var name = "Adam";
```

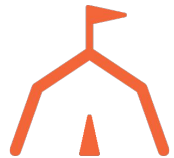


Consider the following code using `const`:

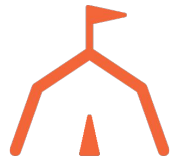
```
const name = "Adam";
console.log(name); // "Adam"
```

With a quick alteration, the code now acts much differently.

```
console.log(name); // ReferenceError: Cannot access 'name'
before initialization
const name = "Adam";
```



Data Types

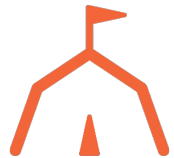


Primitive

A primitive data type is the lowest level of data a language can use. Currently, there are seven primitive data types in JavaScript:

- string
- number
- BigInt
- boolean
- null
- undefined
- Symbol

The other data type in JavaScript is `Object`.



String

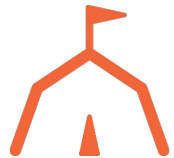
Strings are used to represent text.

Each character in a string references an element.

The first element within a string is considered to be at index 0.

All strings have a length.

```
const framework = "Angular";
console.log(framework.length); // 7
```



Template Strings

When writing more complex strings, template strings provide an excellent solution. Template strings allow for the string to contain embedded expressions and allow for multiline strings.

Rather than using quotations, the backtick (`) character is what begins and ends the template string.

Using template literals produces much cleaner code.

```
const name = "Adam";
console.log(`Hello, my name is ${name}.`); // "Hello, my name is Adam."
console.log(`1 + 1 = ${1 + 1}`); // "1 + 1 = 2";
```

Combining Strings

The addition operator can be used to combine strings.

Using the addition operator is not the preferred way, but it is still beneficial to know.

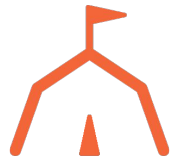
```
const firstName = "Yasmine";
const lastName = "Abdulhamid";
const fullName = firstName + " " + lastName;
console.log(fullName); // "Yasmine Abdulhamid"

let name = "Y";
name += "asmine";
console.log(name); // "Yasmine"
```

Numbers

Numbers represent numbers. There are no differences between integers and floats in JavaScript.

All numbers are just numbers.

```
const firstNumber = 10;
const secondNumber = 20;
const sum = firstNumber + secondNumber;
console.log(sum); // 30
```



BigInt

BigInt is used to represent huge numbers.

BigInt is loosely the same as Number, but for numbers that are outside of the safe limit.

```
const largeNumber = 30n * 39n;
console.log(largeNumber); // 1170n
```

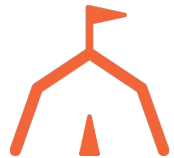
Booleans

Booleans are the "yes" or "no" values in JavaScript.

Instead, they are true or false.

They can only ever be true or false.

```
const isJavaScriptCool = true;
const iAmTired = false;
console.log(isJavaScriptCool); // true;
console.log(iAmTired); // false;
```



Null

Null is the intentional absence of a value.

Null is commonly used as a variable's value if the variable gets initialized later on.

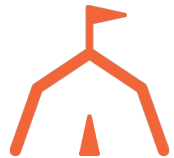
```
let usersName = null;
usersName = "awesomeDude392";
```

Undefined

Undefined is the value every variable is assigned when the script executes.

The value of undefined points to the data not being defined.

```
console.log(food); // undefined
var food = "nachos";
console.log(food); // "nachos"
```



Symbol

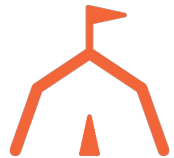
Symbol is a unique and immutable identifier.

Use Symbol as a key to an object.

There isn't a big need to use Symbol early on in development.

Article on Symbol

```
const name = Symbol("myName"); // Symbol(myName)
const age = Symbol("myAge"); // Symbol(myAge)
const myInfo = {
  favoriteLanguage: "JavaScript"
};
myInfo[name] = "Adam";
myInfo[age] = 32;

console.log(myInfo);

console.log(myInfo[name]); // "Adam"
```



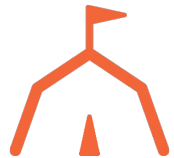
Objects

Objects are collections of key/value pairs (keys are another word for properties).

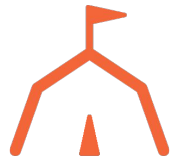
A key must be of type String or Symbol.

The values of keys can be of any data type.

Objects are references to locations in memory - not a defined value.



```
const myInformation = {
  name: "Adam",
  age: 32,
  favoriteFoods: ["nachos", "pizza", "tacos"],
  dog: {
    name: "Syntax",
    breed: "Mix"
  }
};
```



Objects

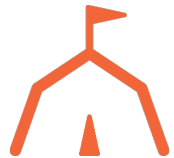
Objects are extremely robust.

Both Function and Array are of type Object.

All primitive types get wrapped in an object wrapper.

The object wrapper is how primitive types gain access to their methods.

Type Checking



Type Checking

As programs develop, there may arise the occasional need to check the type of various data sources.

For instance, if there is a question whether or not a value is of type String or Number, a simple type check could be introduced.

There are two "main" ways of type Checking.

The first is with the `typeof` operator.

The second involves more code but is a much more reliable approach.

Using the typeof operator works nicely if the data is of type Number, String, or Boolean.

```
const language = "JavaScript";
const numberOfStudents = 15;
const isHungry = false;
const assistants = ["Carly", "Ben", "Tim"];
const person = { name: "Yasmine" };
const hello = () => "Hello";

console.log(typeof language); // "string"
console.log(typeof numberOfStudents); // "number"
console.log(typeof isHungry); // "boolean"
console.log(typeof assistants); // "object"
console.log(typeof person); // "object"
console.log(typeof hello); // "function"
```

Using the more involved process gives a much more accurate representation of the data's type.

```
const language = "JavaScript";
const numberOfStudents = 15;
const isHungry = false;
const assistants = ["Carly", "Ben", "Tim"];
const person = { name: "Yasmine" };
const hello = () => "Hello";

function checkType(data) { return
Object.prototype.toString.call(data); }

console.log(checkType(language)); // "[object String]"
console.log(checkType(numberOfStudents)); // "[object Number]"
console.log(checkType(isHungry)); // "[object Boolean]"
console.log(checkType(assistants)); // "[object Array]"
console.log(checkType(person)); // "[object Object]"
console.log(checkType(hello)); // "[object Function]"
```

Arithmetic & String Operators

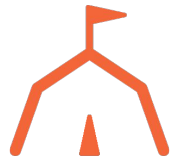
Operator	Purpose	Example
+	Add (numbers)	`let x = y + 3;`
-	Subtract	`let x = y - 7;`
*	Multiply	`let total = price * 5;`
/	Divide	`let sandwiches = slices / 2;`
%	Modulus (remainder)	`let extra = 13 % 5;`
**	Exponent	`let area = side ** 2;`
+	Concatenate (strings)	`let title = "Sir " + name;`

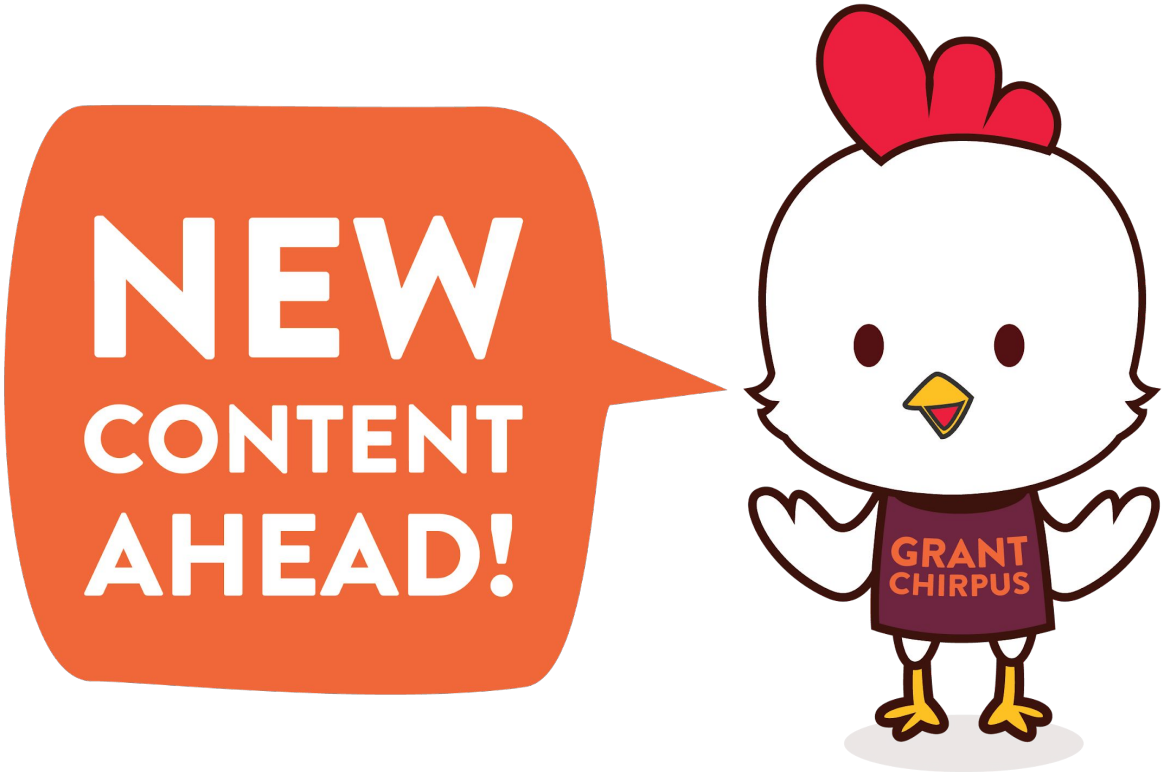
Assignment Operators

Operator	Purpose	Example
=	Set value	`const x = 3;  y = z;`
+=	Add or concatenate and set	`x += 10;` `phrase += "!";`
-=	Subtract and set	`remaining -= 1;`
*=	Multiply and set	`price *= tax;`
/=	Divide and set	`part /= 2;`
%=	Modulus and set	`extra %= 5;`

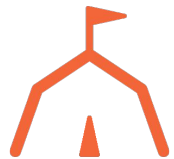
Increment/Decrement Operators

Operator	Purpose	Example
++	Increment (Set value + 1)	`count++;`
--	Decrement (Set value - 1)	`remaining--;`




NEW
CONTENT
AHEAD!
GRANT
CHIRPUS

Control Structures: Conditionals



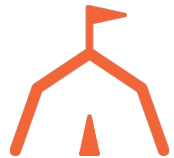
Control Structures

Logic can be inserted to control the flow of programming.

Controlling the flow of a program means programs are structured to perform specific actions under certain conditions.

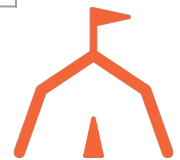
There are a few basic control structures that are important to know:

- Conditionals
- Loops



Any value in JavaScript evaluates to truthy or falsy.

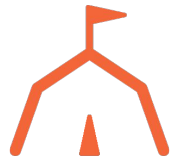
Truthy	Falsy
`true`	`false`
`non-zero numbers`	`null`
`Infinity`	`undefined`
`-Infinity`	`0` (this is the number zero)
`non-empty strings`	`NaN` (“Not a Number”)
`mathematical-expressions`	`empty strings`



If a value is either truthy or falsy, said value could be used to drive logic.

A simple statement to check whether a value is truthy or falsy:

```
const language = "JavaScript";
console.log(!!language); // true
const username = "";
console.log(!!username); // false
```



Comparison Operators

Operator	Purpose	Example
===	Equal (strict)	`num === 360`
!==	Not Equal	`option !== "quit"`
<	Less Than	`price < 10`
>	Greater Than	`score > 50`
<=	Less or Equal	`temperature <= 32`
>=	Greater or Equal	`age >= 21`

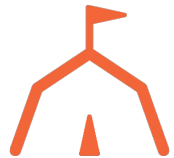
Equals With Coercion (Double Equals)

JavaScript also has older == and != operators that are similar to === and !== but attempt to convert between types to find a match and can give some unexpected results. So most programmers stick with === and !==.

```
3 === "3" // false - === always false when types are different
0 === false // false

3 == "3" // true
0 == false // true - interesting...
true == "true" // false - == not always what you might think
```

Logical Operators

Operator	Purpose	Example
&&	And	`year >= 1900 && year < 2000`
||	Or	`day === "Sat" || day === "Sun"`
!	Not	`!solutionFound` `!( isWeekend || isHoliday )`



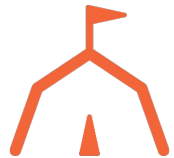
if / else if / else Statements

if / else if / else statements determine which parts of the code should run under certain conditions.

```
if (true) {
  // do something
} else if (true) {
  // do something else
} else {
  // do another something else
}
```

Conditions can be quite simple or quite complex.

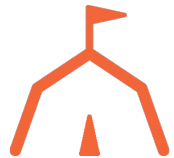
```
let name = "Adam";
let tired = false;
if (name === "Adam" && !tired) {
  console.log("Adam is not tired");
} else {
  console.log("Something is not right...");
}
```



Switch Statements

It is easy to complicate things with the overuse of `else if`.

Switch statements are an excellent alternative.



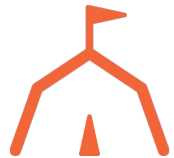
```
const choice = prompt("Select a size");
switch (choice) {
  case "small":
    console.log("You have ordered a small shirt");
    break;
  case "medium":
    console.log("You have ordered a medium shirt");
    break;
  case "large":
    console.log("You have ordered a large shirt");
    break;
  default:
    console.log("We have no other sizes");
    break;
}
```

Ternary Operator

A ternary operator is implemented for very basic decision making.

```
const age = 39;
const ticket = age < 13 ? "child" : "adult";
```

- If age is less than 13, ticket is "child".
- Otherwise, ticket is "adult".



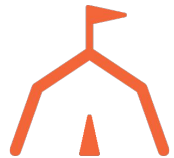
Ternary vs. If Statement

These two snippets do the same thing...

```
let ticket = age < 13 ? "child" : "adult";
```

```
let ticket;
if (age < 13) {
 ticket = "child";
} else {
 ticket = "adult";
}
```

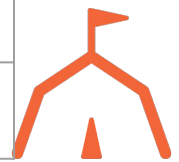
Control Structures: Loops



Loops

Loops allow code blocks to repeat over and over again.

Loop	Use
`for`	Repeatedly executes a block of code using a counter variable.
`do...while`	Repeats until the condition evaluates to false. Always executes at least once.
`while`	Repeats until the condition evaluates to false. May not execute at all if condition starts out false.
`for...in`	Iterates over keys of an object (including arrays).
`for...of`	Iterates over values of an object (including arrays).



Loops

Control structures have statements that perform specific actions when used in a loop.

`break`: Used to break out of a control structure.

`continue`: Used to skip to the next iteration.

for Loops

The `for` loop has three optional expressions, enclosed in parentheses and separated by semicolons, followed by a statement or a set of statements executed in the loop.

```
for (initialization; condition; final-expression) {
  // repeating things
}

for (let i = 0; i <= 10; i++) {
  // console.log(i);
}
```

while Loops

The `while` loop iterates as long as a given condition evaluates to true. The condition evaluates before each iteration of the loop.

```
while (condition) {
  // repeating things
}

let a = 0, j = 0;
while (a < 30) {
  a++;
  j += a;
}
```

do...while Loops

The `do...while` loop is very similar to a while loop. The code contained within the code block runs at least once before the condition evaluates.

```
do {
  // repeating things
}
while (condition);

let userPass = null;
do {
  userPass = prompt("What is your password?");
} while (userPass !== "secret");
```

for...of Loops

The `for...of` loop iterates through property values.

It is quite simple to set up and acts very similar to the `for...in` loop.

The `for...of` loop should be your go-to for arrays.

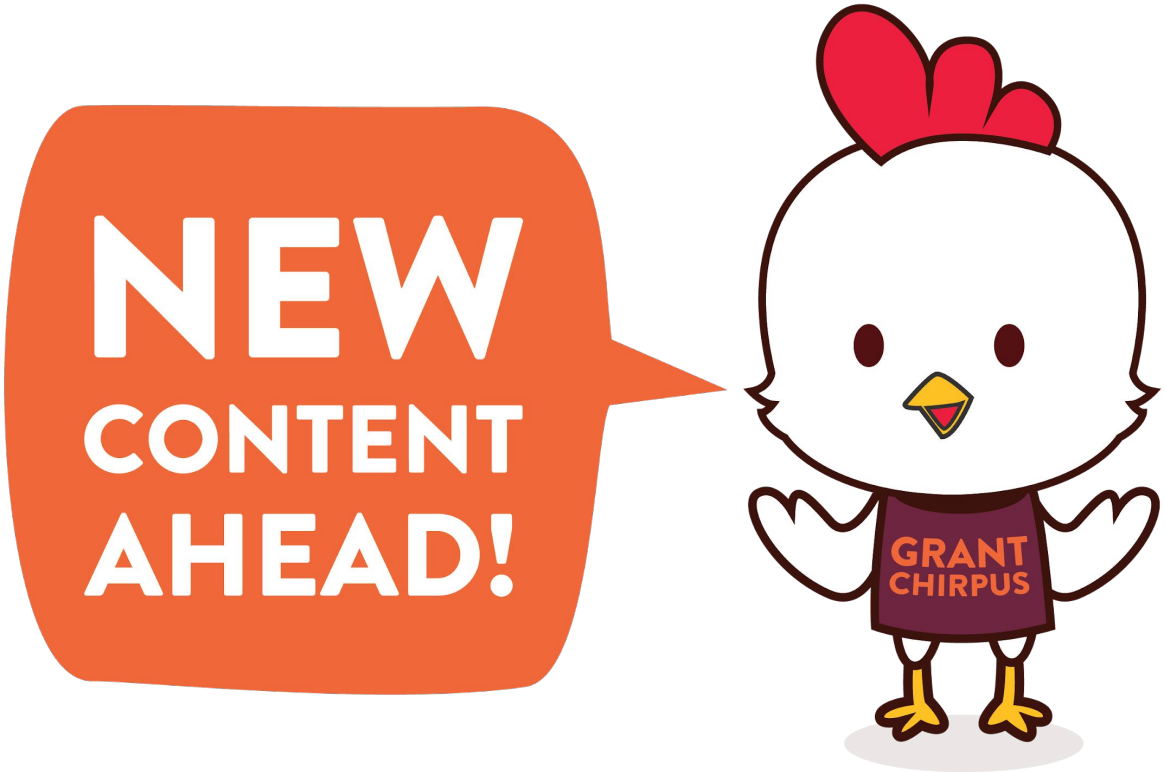
```
let languages = ["Java", "JavaScript", "Ruby",
"Python", "C#", "PHP", "HTML", "CSS"];
for (let language of languages) {
  console.log(language);
}
```

for...in Loops

The `for...in` loop iterates through property names.

The `for...in` loop is quite handy for checking properties of an object or extracting those property names.

```
const grandCircus = { location: "Detroit", rooms: 6, colors: ["teal", "orange", "charcoal"] };
for (let prop in grandCircus) {
  console.log(prop);
}
```


NEW
CONTENT
AHEAD!
GRANT
CHIRPUS

Functions

Functions

Executing a code block on demand is a widespread task in programming.

Doing such a task would either require the code to be written multiple times or a function to be implemented.

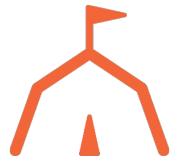
Functions, by default, return a value.

```
function greetClass() {
  console.log("Hello class!!!");
}

greetClass(); // "Hello Class!!!" is logged, however, the
function returns undefined
```

return Statement

The `return` statement causes the function to immediately exit and returns the value to the line of code that called the function.

No code is executed after a return statement so the return statement should always be the last line of code within a function.



Types of Functions

With ES6, there are four "types" of functions.

`function declaration`

`function expression`

`arrow function expression`

`anonymous function`

All four require the same fundamental knowledge of functions.

Function Declaration

A function declaration is the declaration of a function.

Function declarations do get hoisted.

Declaring a function is relatively similar to declaring variables (keyword, name).

```
function calculateArea(length, width) {
  console.log(length * width);
  return length * width;
}

calculateArea(10, 5); // 50 is logged, however, the function
returns the number 50
```

Function Expression

Similar to the function declaration; however, the big difference is that the function is the value of a variable.

The function is also not named, and it is anonymous.

```
const calculateArea = function(length, width) {
  return length * width;
}

calculateArea(10, 5); // the function returns the
number 50
```

Arrow Function

Arrow functions are quite similar to function expressions but require slightly less code, and arrow functions handle referencing `this` differently.

`this` is a word that comes back up later in the course.

```
// function declaration
function feedPeople(person1, person2) {
  return `${person1} and ${person2} are now fed.`;
}

// arrow function
const feedPeople = (person1, person2) => `${person1} and
${person2} are now fed.`;
```

Parameters

When a function needs any information to complete its task, `parameters` get defined within the declaration.

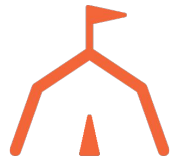
Parameters act like variables; however, they are not indeed variables.

Parameters are just placeholders.

Arguments

When a function is called or executed, data may get passed into the function as `arguments`.

The `arguments` used become assigned to the parameters for use within the function.



Example

Let us take a few minutes to discuss what is happening.

`number` is the parameter.

`4` is the argument.

```
function isEven(number) {
  if (number % 2 === 0) {
    return true;
  } else {
    return false;
  }
}

isEven(4);
```

Default Parameters

Parameters may also have default values.

If the function executes with an undefined argument, the default parameter kicks into play.

```
const feedPeople = (person1, person2 = "Carol")
=> `${person1} and ${person2} are now fed.`;

feedPeople("Steve"); // "Carol" is assigned to
person2
```

Rest Parameter

The purpose of the rest parameter is to bundle up all of the remaining arguments of the function in an array (a JavaScript list of values).

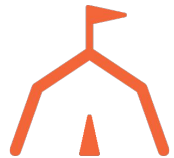
It is specified using three dots.

```
function myFunction(param1, param2, ...others) {
  // code here
}
```

Example

This function can only ever use two parameters (num1 and num2) even if the function is called with three arguments.

```
function addNumbers(num1, num2) {
  let total = 0;
  total = num1 + num2;
  return total;
}
addNumbers(1, 2, 3, 4, 5); // returns 3
```



Example

This function, using the rest parameter and reduce, allows the function to use any amount of arguments.

```
function addNumbers(...numbers) {
  return numbers.reduce((prev, next) => prev +
next);
}
addNumbers(1, 2); // returns 3
addNumbers(1, 2, 3, 4, 5); // returns 15
```

Rest Parameter

The rest parameter must be the last parameter assigned within the function.

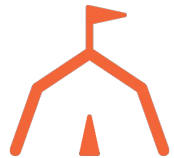
The "rest" of the arguments get bundled in an array.

```
function sayHello(person1, person2, ...everyoneElse) {
  console.log(`Hello ${person1}`); // Hello Adam
  console.log(`Hello ${person2}`); // Hello Ivan
  console.log(`Hello ${everyoneElse[2]}`); // Hello Celena
  console.log(`Hello ${everyoneElse[3]}`); // Hello Yasmine
}

sayHello("Adam", "Ivan", "Kim", "Antonella", "Celena",
"Yasmine");
```

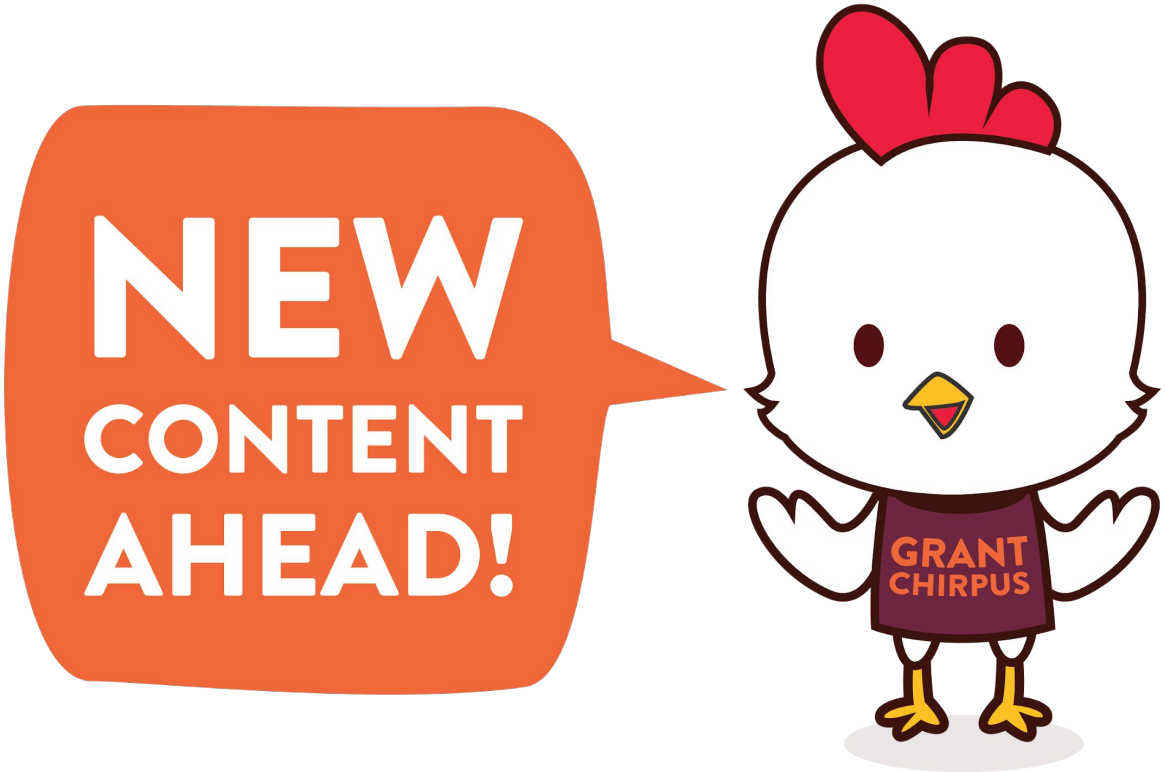
Return Functions

Functions can return functions.

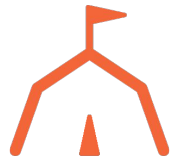
Returning functions from functions is a common practice.



```
function success() {
  console.log("The number is even.");
}

function isEven(num) {
  if (num % 2 === 0) {
    return success;
  } else {
    return function() {
      console.log("The number is not even.");
    }
  }
}

let result = isEven(4); // value of result is now the success
function

result(); // result is now a function and can be executed to see "The
number is even."
```


NEW
CONTENT
AHEAD!
GRANT
CHIRPUS

Objects



Objects

Objects are a data structure that allows us to store unorganized collections of data.

An object consists of two parts: state and behavior. That is, data about the object (state) and methods that can act on that data (behavior).

There are a few ways to create an object.

The first option for creating an object is object literal notation.

The second way is using a constructor.

Objects

Objects require curly braces (unlike arrays, which use square brackets).

Objects have key/value pairs (properties/values), separated by a colon.

If there are multiple properties, a comma separates them.

```
const car = {
  make: "Ford",
  model: "Ranger",
  year: 2019,
  color: "blue"
};
```

Object keys can be accessed or altered using dot or bracket notation.

Adding properties is done by either dot or bracket notation.

Deleting properties can be done by using the `delete` keyword.

```
const car = { make: "Ford", model: "Ranger", year: 2019,
color: "blue" };
console.log( car["make"] ); // "Ford" is logged
console.log( car.model ); // "Ranger" is logged

car.ecoFriendly = true; // add ecoFriendly: true

delete car.color; // car no longer has a color key
```

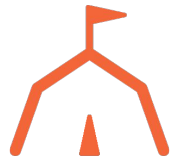
Methods

Methods are very similar to functions but have a more specific purpose.

Methods belong to objects which allow the objects to perform actions.

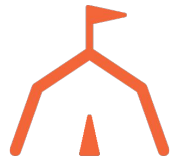
Often, methods will be used to do the following:

- Return some information about the object
- Change the state of the object



```
const car = {
  make: "Ford",
  model: "Ranger",
  year: 2019,
  color: "blue",
  describe() {
    console.log(`${this.year} ${this.make} ${this.model}`);
  }
  getYear() {
    return this.year;
  }
  paintTheCar(color) {
    this.color = color;
    console.log(`The car is now ${this.color}.`);
  }
};
car.describe();
console.log( car.getYear() );
car.paintTheCar("orange");
```

Arrays



Arrays are similar to objects in that they are collections of data.

An array is a data structure consisting of an ordered collection of elements.

This numerical index starts at 0, not 1.

```
const names = ["Adam", "Anna", "David", "Josh", "Yasmine"];
// Index:        0        1       2        3         4
```

Similar to objects, arrays can be defined through literal syntax or a constructor.

The literal syntax is the preferred way of doing things, so stick with that.

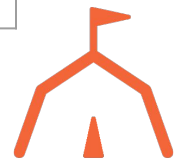
Bracket notation is used to access the value in an index.

```
const people = ["Adam", "Amanda", "Ashley",
"Ajay", "Jonah"];
const firstPerson = people[0]; // "Adam"
const lastPerson = people[people.length - 1]; //
"Jonah"
```

Adding to an Array

The following methods allow for the adding of elements to an array:

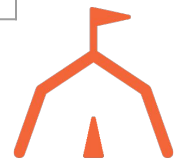
Method	What the method does
`Array.unshift()`	Adds the element to the beginning of the array
`Array.push()`	Adds the element to the end of the array
`Array.splice()`	Adds the element to a specific index in the array



Removing from an Array

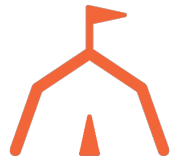
The following methods allow for the removing elements from an array:

Method	What the method does
`Array.shift()`	Removes the element at the beginning of the array
`Array.pop()`	Removes the element at the end of the array
`Array.splice()`	Removes element(s) at a specific index in the array



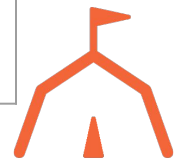
A method that can both add and remove is Array.splice().

```
const drinks = ["Tea", "Coffee", "Soda",
"Water"];
drinks.splice(2, 1, "Pop"); // start at index 2,
remove 1 element, add "Pop"
console.log(drinks); // ["Tea", "Coffee", "Pop",
"Water"];
```



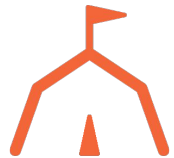
Other Array Methods

Method	What the method does
`Array.includes(value)`	Returns true if the value is in the array; false if not.
`Array.indexOf(value)`	Returns the index of the value in the array, or -1 if not found.
`Array.findIndex(fn)`	Returns the index where the callback function finds a match.
`Array.filter(fn)`	Returns a new array that only has the elements that match the callback function.

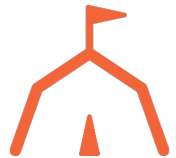


Array Reference

- MDN has documentation for arrays and list all the methods you can use. (There are a lot more that we didn't cover.)
- W3Schools also has an introduction to arrays and a reference list of methods.



Spread



Spread

The spread operator is used to extract elements of an array or the keys of an object.

Think about the following situations:

- An array/object is needing a copy
- A function requires multiple arguments from the same object/array

Encountering each of the above situations is very likely.

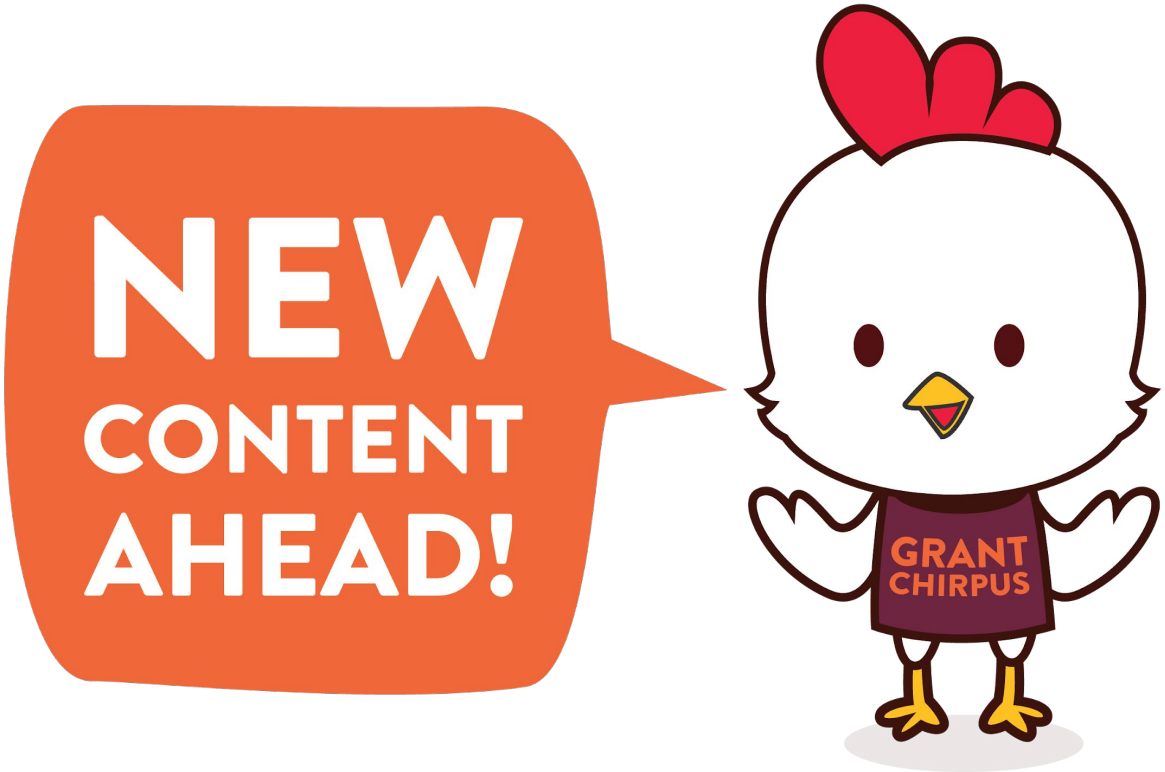
Copying an object to manipulate the copied version, not the original, is quite common.

In this situation, it is quite common to construct an object literal from the original object's properties.

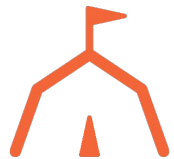
```
const obj1 = { name: "Snoopy", age: 2 };
const obj2 = { name: obj1.name, age: obj1.age };
const obj3 = { ...obj1 }; // obj3 contains the
same keys/values as obj1

const names1 = ["Adam", "Grace"];
const names2 = [ ...names1, "Cody" ]; // names2
contains the same elements as names1 and "Cody"
```

The spread operator can also be used as a single argument that refers to a collection of elements.

```
const things = ["Computer", "TV", "Radio", "Cellphone"];
function printThings(item1, item2, item3, item4) {
  console.log(item1);
  console.log(item2);
  console.log(item3);
  console.log(item4);
}
printThings(...things);
```


NEW
CONTENT
AHEAD!
GRANT
CHIRPUS

Scope & Closures



Declaring Variables

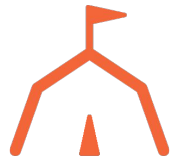
A variable declared within a function is scoped to that function.

Likewise, declaring a variable with the `let` or `const` keyword, the variable is only accessible within that code block.

Global Variables

Declaring a variable or function anywhere out of a code block establishes said variable or function as global.

Global variables and functions can be accessed anywhere but can cause problems with larger, more complex applications.



```
const dayOfWeek = "Monday"; // scoped globally
function processUser() {
  // newUser is scoped to processUser
  const newUser = "Adam";
  console.log(newUser); // the string "Adam"
  console.log(dayOfWeek); // the string "Monday"
}

processUser();
console.log(dayOfWeek); // the string "Monday"
console.log(newUser); // ERROR: doesn't exist
outside of processUser
```

IIFE

A significant problem in JavaScript is the global scope and keeping it clean.

As such, several critical techniques and design patterns in JavaScript were developed to address this problem.

One such technique is the **Immediately Invoked Function Expression (IIFE)**.

IIFE

An IIFE is an anonymous function that executes immediately.

The IIFE creates a *closure* that keeps all the variables inside private and contained.

```
(function() {
  // awesome code
})();

"use strict";
{
  // the new ES6 block can also be used to keep variables scoped
properly!
}
```

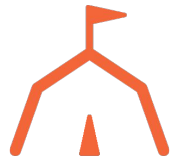
Closures

What are closures?

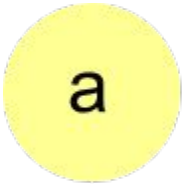
A closure is a function that has references to variables and functions that are coming from an outer scope.

As a nifty side effect, developers can create closures as a form of encapsulation in JavaScript.

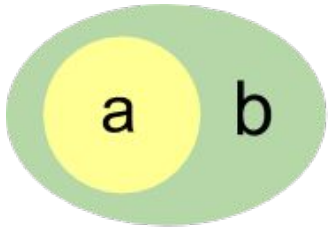
```
function yellow() {
  const a = "yellow";
  function green() {
    const b = "green";
    console.log("a: " + a); // > a: yellow
    console.log("b: " + b); // > b: green
  }
  green();
}
yellow();
```

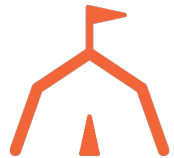


The set of variables set to yellow:



The set of variables set to green:





Closures

Why are closures so powerful?

Programmers can create numerous functions that all run the same base code, but have different values associated with them.

```
function sayHello(name) {
  const greeting = function(message) {
    return `${message} ${name}?`;
  }
  return greeting;
}

let hiAdam = sayHello("Adam");
console.log(hiAdam("How are you"));

let hiYasmine = sayHello("Yasmine");
console.log(hiYasmine("How are you"));

let hiDavid = sayHello("David");
console.log(hiDavid("How are you"));
```

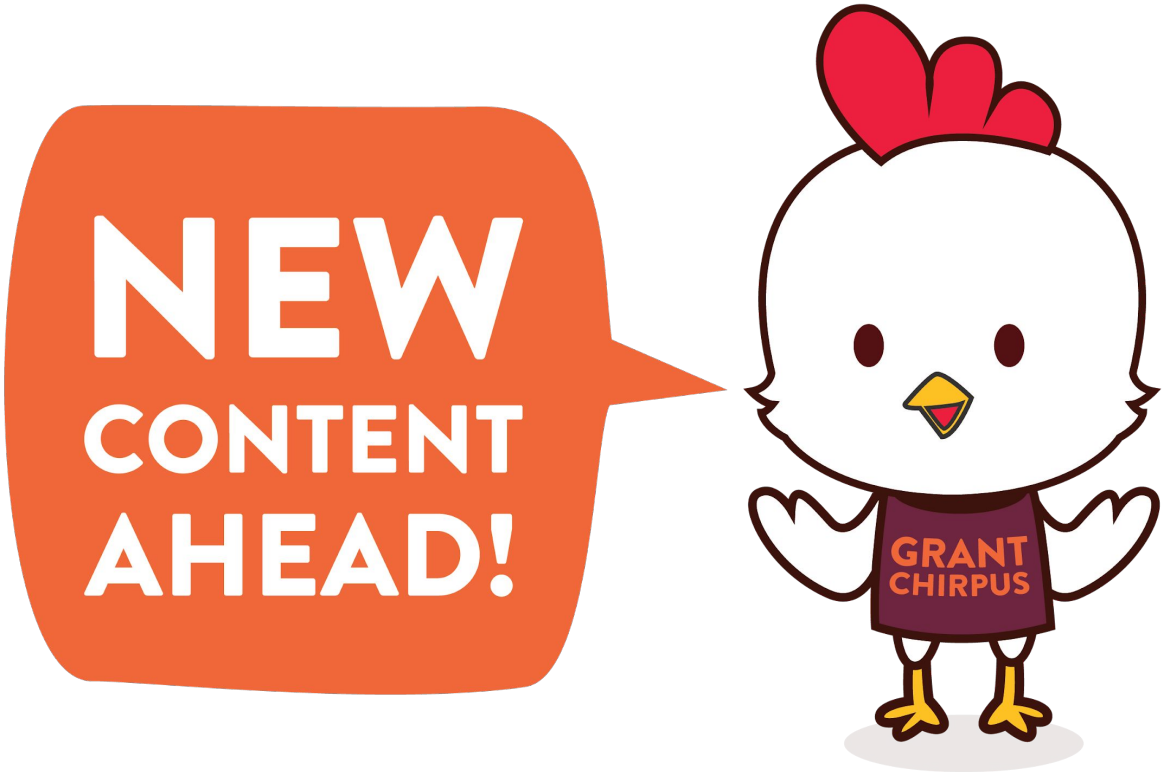
Closures

Variables inside a closure are only accessible within that closure.

In JavaScript, closure is very close to the same concept as a function's scope.

A closure "remembers" the context in which it is declared.

Even if that function is removed from that context and used elsewhere, it still has access to all the variables of its original context.


NEW
CONTENT
AHEAD!
GRANT
CHIRPUS

Classes

Classes

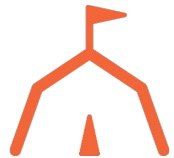
Classes allow for a clean and easy way to create objects.

Behind the scenes, a class is a function.

Think of a class as a blueprint or model.

Objects are instantiated (created) from the blueprint or model.

Class declarations are *not* hoisted.

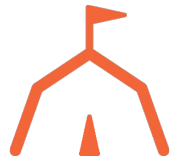


For Example

Imagine writing a program to manage all students for bootcamp.

There would probably be a single class (blueprint), which acts as the structure for each student (object).

Each student (object) may have unique attributes, but they are all derived from the same structure (class).



The object is constructed with the provided state.

```
class Student {
  constructor(name, age){
    this.name = name;
    this.age = age;
  }
}

const tSwift = new Student("Taylor Swift", 30);
console.log(tSwift); {name: "Taylor Swift", age: 30}
```

Adding behavior to the object requires writing a method to the class.

```
class Student {
  constructor(name, age){
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log(`Hello, I'm ${this.name}`!`);
  }
}

const tSwift = new Student("Taylor Swift", 30);
tSwift.sayHello(); // Hello, I'm Taylor Swift!
```

`this` always refers to a single object.

The object `this` references is determined by when and where it gets used.

In the global context, `this` refers to the global window object.

When using `this` within a method, it refers to that particular object.

```
const javaScriptStudent = {
  name: "Stephanie",
  age: 30,
  sayName() {
    console.log(`My name is ${this.name}`);
  }
};
javaScriptStudent.sayName(); // "My name is Stephanie"
```

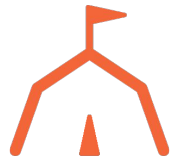
Prototype

Inspecting these objects after creation will not display the methods.

The object's prototype contains the methods.

The **prototype** holds all the common properties and methods (push, pop, splice, and the other pre-defined methods for an array).

For example: when an array is defined, it may have unique elements but also inherits conventional methods and properties from the array's prototype.



Prototype

each object has its own values

object

- name: Taylor Swift
- age: 30

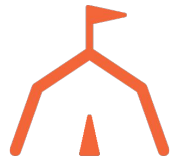
object

- name: Grant C.
- age: 8

prototype

- sayHello: *function*

prototype holds shared values

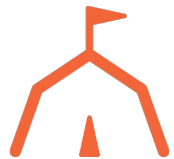


hasOwnProperty

The `hasOwnProperty` method can determine whether a property is part of the individual object or part of its common class.

```
console.log(tSwift.hasOwnProperty("name")); // true
console.log(tSwift.hasOwnProperty("sayHello")); // false

let newArray = new Array();
console.log(newArray.length);
console.log(newArray.pop);
console.log(newArray.hasOwnProperty("length")); // true
(individual object)
console.log(newArray.hasOwnProperty("pop")); // false (common
class)
```

Common Methods



Common Methods

Commonly used methods

There are many methods provided by JavaScript.

There is hardly a need to memorize the list of methods or trying to understand precisely how they all work.

Focus on knowing how to look up methods and use them.

Method List