Before we begin...

- Videos On!

Welcome

Agenda

- Review
- Comparison Operators
- Conditionals
- Logical Operators
- Advanced Conditionals (maybe)
- Loops (maybe)
- Arrays (maybe)
- Recap

Review

- Web Development
- JavaScript's Role
- Data Types
- Variables

Comparison Operators

What are comparison operators?

- Comparison operators allow us to compare two values
- They always *return a boolean value*
- There are lots of them!

==

Often called the "loose equality" operator, this will *compare two values* and **return a boolean**

It'll try to make the types the same using *type coercion*

```
5 == 5; // true

"First string" == "Second string"; // false

2 == "2"; // true
```

===

Often called the "strict equality" operator (or threequal), this will *compare two values* and **return a boolean**

It cares about both the value and the type and *will not use type coercion*

```
26 === 26; // true

"String" === "String"; // true

"16" === 16; // false
```

My (biased) advice?

- Always use the strict equality operator (===)
- Explicitly work with your types
  - Make sure you know what type things are
  - Convert them when necessary (e.g. `parseInt` etc.)

!=

!=

Often called the "loose inequality" operator, this will *compare two values* and **return a boolean**

It *will use type coercion*, if necessary

```
25 != 24; // true

"String" != "string"; // true

24 != 24; // false

19 != "19"; // false
```

!==

!==

Often called the "strict inequality" operator, this will *compare two values* and **return a boolean**

It *will not use type coercion*, as it cares about both value and type

```
11 !== 13; // true

"Hello" !== "hello"; // true

38 !== "38"; // true

24 !== 24; // false
```

<

The less than operator, it'll always *return a boolean*

```
4 < 10; // true

15.6 < 21.94; // true

25 < 25; // false

32 < 16; // false
```

<=

<=

The less than or equals to operator, it'll always *return a boolean*

```
11 <= 15; // true

13.1 <= 23.94; // true

25 <= 25; // true

31 <= 19; // false
```

>

The greater than operator, it'll always *return a boolean*

```
24.124 > 12.522; // true

32 > 16; // true

15.6 > 21.94; // false

25 > 25; // false
```

>=

>=

The greater than or equals to operator, it'll always *return a boolean*

```
25 >= 25; // true

31 >= 19; // true

11 >= 15; // false

13.1 >= 23.94; // false
```

Conditionals

The Typical Execution of a Script

JavaScript is an asynchronous programming language.

It runs line-by-line from the top, and starts the execution of each line in order (but *it won't wait for the completion of each line!*)

What are conditionals?

Conditionals are the way that we make decisions with code. It is a way to decide what lines execute and what lines to skip, based on the value of some expression (meaning that it relies on boolean*ish* values)

We use them to add logic, to make decisions, to allow data to change how our script runs and to make parts of our code optional

Let's have a think...

- How could we determine if a person can legally get their driver's license?
- How could we determine if a person should be logged in to their account?
- How could we determine if a given programming language is a front-end programming language?
- How could we determine if a year is in the 20th Century (1901 - 2000)?
- How could we determine if someone should be able to purchase something?

```
if
```

The `if` statement

The fundamental control statement that allows JavaScript to make decisions

It is made up of the `if` keyword, parentheses wrapping a condition and curly brackets to identify what code should execute if the condition is truthy

```
if (true) {
  // Code to execute
}
```

The `if` statement

```
if (2 === 2) {
  // Maths seems to be working
}

let courseContent = "JS";

if (courseContent === "JS") {
  // This is going to be fun!
}

if (courseContent === "Ichthyology") {
  // This will be skipped!
}
```

if … else

The `if ... else` statement

```
if (CONDITION) {
  // Code to execute if the condition is truthy
} else {
  // Code to execute if the condition is falsey
}
```

The `if ... else` statement

```
if (2 === 2) {
  // Maths seems to be working
} else {
  // That's a real worry...
}
```

```
let number = 14;

if (number > 0) {
  // The number is positive
} else {
  // The number is negative
}
```

if … else if … else

The `if ... else if ... else` statement

You can have as many `else if`s as you need

JavaScript *will only run the code in the first passing condition's block* (curly brackets). It will skip the rest!

```
if (FIRST_CONDITION) {
  // Code to execute if the FIRST_CONDITION is truthy
} else if (SECOND_CONDITION) {
  // Code to execute if the SECOND_CONDITION is truthy
} else {
  // Code to execute if both conditions are falsey
}
```

The `if ... else if ... else` statement

```
let role = "Lead Instructor";

if (role === "Lead Instructor") {
  // Code to execute
} else if (role === "Instructional Associate") {
  // Code to execute
} else {
  // Code to execute
}
```

Logical Operators

What are Logical Operators?

Logical operators are used to make more complex decisions or conditionals. They can also be used to combine multiple conditionals.

For example:

- `CONDITION_ONE` needs to be true **AND** `CONDITION_TWO` needs to be true
- `CONDITION_ONE` needs to be true **OR** `CONDITION_TWO` needs to be true
- `CONDITION_ONE` must **NOT** be true

|| (OR)

The logical **OR** operator is represented by two pipe symbols (the vertical lines often above the enter/return key). As long as one item is truthy, it will pass (meaning *return true*)

```
true || true; // true

true || false; // true

false || true; // true

false || false; // false
```

|| (OR)

```
let lang = "JS";

if (lang === "HTML" || lang === "CSS" || lang === "JS") {
  // You are talking about a front-end language
} else {
  // It is probably a back-end language
}
```

How does | | work?

It reads from left to right, and it will evaluate each operand and turn it into a boolean value. If the result is true at any point, it stops and will return that operand. If it has gone through all the operands and nothing returns true, it will return the last operand

More or less, it'll return the first truthy value it finds, or the last value

&&

&& (AND)

The logical AND operator is represented by two ampersands. Both operands (the things on both sides) need to return true for the entire expression to return true

```
true && true; // true

true && false; // false

false && true; // false

false && false; // false
```

&& (AND)

```
let usedLoggedIn = true;
let userIsAdmin = true;

if (userLoggedIn === true && userIsAdmin === true) {
  // You have complete control
}

// Question: Could we shorten that conditional?
```

&& (AND)

```
let validCreditCardDetails = true;
let balance = 100;
let itemCost = 80;

if (validCreditCardDetails && balance >= itemCost) {
  // Yes, you can purchase this item
}
```

How does && work?

It reads from left to right, and it will convert each operand into a boolean. If that conversion results in false, the whole thing will stop and it will return the falsey boolean. If all operands were truthy, it'll return the last operand

More or less, it returns the first falsey value it finds or the last value if everything was truthy

! (NOT)

The logical NOT operator is represented by the exclamation point. It accepts a single argument, converts it to a boolean and then returns the opposite of that boolean

```
!true; // false

!false; // true

!"Hello"; // false

!42; // false

!0; // true
```

! (NOT)

```
!!true; // What would this return?

!!"Hello"; // What would this return?
```

! (NOT)

```
let hasAccount = false;

if (!hasAccount) {
  // You can create an account
} else {
  // You already have an account
}
```

Start the exercises here, please!

See you in 10 minutes

Loops

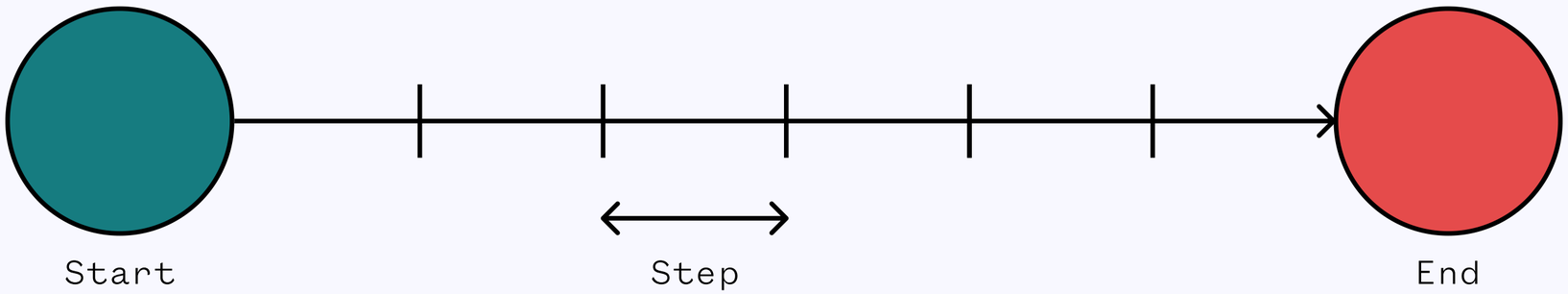
What are loops?

A loop is a block of code that can execute over and over again.

It consists of three must-have parts:

- A starting point
- An ending condition
- An increment, or a step

You get access to the current value (e.g. how far it has gone in the loop)


Start
Step
End

Start, Step, End

Think of it as a running race. You start, you keep stepping until you reach 100 metres and then you stop

The `while` loop

The `while` loop

```
let count = 0; // Start

// End Condition
while (count < 10) {
  // Do something
  count = count + 1; // Step
}
```

The `for` loop

The `for` loop

```
for (let count = 1; count <= 10; count = count + 1) {
  // Do something
  console.log(count);
}

// Think of it as:
// for (START; END; STEP) {}
```

My (biased) advice

Stick to using the `for` loop in the beginning, as it ensures you have all the necessary parts and all of the important things are in one, consistent place

Double check all of your loops before you run them! Otherwise, you run the risk of an infinite loop

Start the exercises here, please!

See you in 10 minutes

Structural Data Types

What are Structural Data Types?

- They are structures that JavaScript has to *organise data*
- They can contain other data types (both primitive and structural data types)
- They have *distinguishable parts* (e.g. a first item, a key-value pair etc.)
- They are also called Composite Data Types

What Structural Data Types does JavaScript have?

Well, technically it only has two:

- Objects (which includes things like Arrays, Dates, Maps, and Sets)
- Functions

Arrays, Objects and Functions will be the ones we focus on though

Our First Structural Data Types

In JavaScript, when people think of structural types - two main ones come to mind.

Arrays: Ordered data that you access with *an index*

Objects: Unordered data that you access with a *key*

Our First Structural Data Types

```
let myArray = ["A", "B", "C"];

let myObject = {
  firstKey: "firstValue",
  secondKey: "secondValue",
};
```

Arrays

What are Arrays?

Arrays are lists that can be filled with any data type

They are **ordered** and you access data with **a zero-based index**

They are able to be iterated through (meaning looped through)

What is an index?

An index is a number that allows us to **access items in an Array** (or similar collection)

It is **zero-based**, meaning the first item in the Array is index 0, the second item is index 1, and so on

Creating Arrays

```
let emptyArray = [];

let randomNumbers = [12, 42, 1, 3, 92];

let dataTypes = [true, null, 14, "string"];
```

```
let weirdInstruments = [
  "The Great Stalacpipe Organ",
  "Stylophone",
  "Ondes Martenot",
  "Sharpischord",
  "Hydraulophone",
  "Pyrophone",
];
```

Accessing Elements

```
let letters = ["A", "B", "C"];

letters[0]; // "A"
letters[1]; // "B"
letters[2]; // "C"
letters[3]; // What would this return?
```

Reassigning Elements

```
let instruments = [
  "The Great Stalacpipe Organ",
  "Stylophone",
  "Ondes Martenot",
  "Sharpischord",
  "Hydraulophone",
  "Pyrophone",
];

weirdInstruments[0] = "Roli Seaboard";
weirdInstruments[3] = "Makey Makey Banana Piano";
```

Getting the `.length`

```
let instruments = [
  "The Great Stalacpipe Organ",
  "Stylophone",
  "Ondes Martenot",
  "Sharpischord",
  "Hydraulophone",
  "Pyrophone",
];

instruments.length; // 6

// What will the following lines return?
instruments[instruments.length];
instruments[instruments.length - 1];
```

Arrays are very consistent

```
let ordinals = ["Zeroeth", "First", "Second", "Third"];

ordinals[0]; // "Zeroeth"
ordinals[1]; // "First"
ordinals[2]; // "Second"
ordinals[3]; // "Third"

// Very consistent...
// Maybe we could generate these indexes with a loop!
```

Looping through Arrays

```
let ordinals = ["Zeroeth", "First", "Second", "Third"];
```

If we were to loop through this array:

- What is our starting point?
- What is our ending condition?
- What is our step/increment?

Looping through Arrays

```
let ordinals = ["Zeroeth", "First", "Second", "Third"];

for (let index = 0; index <= 3; index += 1) {
  let currentElement = ordinals[index];
  console.log(currentElement);
}

// What would happen if we added or removed items?
```

Looping through Arrays

```
let ordinals = ["Zeroeth", "First", "Second", "Third", "Fourth"];

for (let index = 0; index < ordinals.length; index += 1) {
  let currentElement = ordinals[index];
  console.log(currentElement);
}
```

Properties and Methods

```
let ordinals = ["First", "Second", "Third"];

ordinals.length; // 3

ordinals.push("Fourth"); // Add "Fourth" to the end
ordinals.pop(); // Remove the last element ("Fourth")

ordinals.unshift("Zeroeth"); // Add "Zeroeth" to the start
ordinals.shift(); // Remove the first element ("Zeroeth")

ordinals.indexOf("Second"); // 1
```

Start the exercises here, please!

See you in 10 minutes

Objects

What are Objects?

- Objects are a collection of key-value pairs (like a word and a definition in a dictionary)
- They are **unordered**
- They, like Arrays, can store any combination of data types

Why use Objects?

- They allow us to effectively interact with large amounts of data
- They allow for encapsulation and modularity:
  - It is a way to group functionality and data
  - It makes sharing your code a lot easier

Creating Objects

```
let emptyObject = {};

let movie = {
  name: "Satantango",
  director: "Bela Tarr",
  duration: 432,
};
```

Creating Objects

```
let bookSeries = {
  name: "In Search of Lost Time",
  author: "Marcel Proust",
  rating: 5,
  books: [
    "Swann's Way",
    "In the Shadow of Young Girls in Flower",
    "The Guermantes Way",
    "Sodom and Gomorrah",
    "The Prisoner",
    "The Fugitive",
    "Time Regained",
  ],
};
```

Accessing Values

```
let bookSeries = {
  name: "In Search of Lost Time",
  author: "Marcel Proust",
  rating: 5,
  books: ["Swann's Way"],
};

let name = bookSeries.name; // "In Search of Lost Time"
let author = bookSeries.author; // "Marcel Proust"
let rating = bookSeries.rating; // 5

let firstBook = bookSeries.books[0]; // "Swann's Way"
```

Changing Values

```
let movie = {
  name: "Satantango",
  director: "Bela Tarr",
  duration: 432,
};

movie.name = "Sátántangó";
movie.director = "Béla Tarr";
movie.duration = 534;
```

Adding new Values

```
let movie = {
  name: "Satantango",
  director: "Bela Tarr",
  duration: 432,
};

movie.language = "Hungarian";
movie.ratingOutOfFive = 10;
movie.parts = 12;
```

Nested Objects

```
let explorer = {
  firstName: "Jacques",
  lastName: "Cousteau",
  birth: {
    day: 11,
    month: 6,
    year: 1910,
  },
};

let firstName = explorer.firstName;
let birthDay = explorer.birth.day;
let birthMonth = explorer.birth.month;
let birthYear = explorer.birth.year;
```

Complex Data Structures

```
let marxFamily = [
  { name: "Groucho", birthYear: 1890 },
  { name: "Harpo", birthYear: 1888 },
  { name: "Chico", birthYear: 1887 },
  { name: "Zeppo", birthYear: 1901 },
  { name: "Gummo", birthYear: 1893 },
];

for (let i = 0; i < marxFamily.length; i += 1) {
  let brother = marxFamily[i];
  console.log(brother.name, brother.birthYear);
}
```

Start the exercises here, please!

See you in 10 minutes

Let's have a think...

- What if we wanted a piece of code to execute every time something occurred (e.g. a click, or a keypress)?
- What if we wanted a piece of code to execute every second?
- What if we wanted to perform a really complicated task over and over again?

Functions

What are Functions?

The main building blocks of programs - they are reusable sections of code (almost like subprograms)

They are very useful for when we need to perform a single action in many places of the script, and for reducing repetition

What are Functions?

We give a name to a part of our program, and in doing so, we make it flexible, reusable and more readable

Functions are also the way we execute code based on events (e.g. clicking, scrolling etc.)

How do they work?

- We **define** a function
- We **call** (or **execute**) it when we want the code within the function to run

Why do we need Functions?

- Because sometimes we need to do things a lot of times, or at regular intervals
- Because sometimes we need to react to certain events taking place (e.g. a click)
- Because sometimes we need our code to be dynamic (that's where parameters and arguments come into play - you'll see them soon)

What can Functions do?

- Calculations
- Animations
- Change CSS
- Change, add or delete elements on the page
- Speak to a server (e.g. an API)
- Anything!

Declaring Functions

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

```
function add() {
  console.log(2 + 2);
}

function subtract() {
  console.log(10 - 3);
}
```

Declaring Functions

```
// You will also see these two variations!

// Function Expression
let speak = function () {
  console.log("Hello");
};

// Arrow Function
let add = () => {
  console.log(2 + 2);
};
```

Calling Functions

```
// Function Declaration

function sayHello() {
  console.log("Hello");
}

// Function Call Site

sayHello();
```

Parameters and Arguments

Our functions aren't dynamic just yet. This is where we need to see **parameters** and **arguments**

They are how we provide a function with extra data or information

Parameters and Arguments

We listen for **parameters** in the function declaration

We provide **arguments** at the call site

Parameters and Arguments

```
// `name` is our parameter

function sayHello(name) {
  let greeting = "Hello " + name;
  console.log(greeting);
}

// "Jacques" is our argument

sayHello("Jacques");
```

Parameters and Arguments

```
function add(numOne, numTwo) {
  let solution = numOne + numTwo;
  console.log(solution);
}

add(5, 10);
// numOne will be set to 5, numTwo will be set to 10

add(3, 18);
// numOne will be set to 3, numTwo will be set to 18
```

`return` Values

Sometimes your function calculates something and you want the result!

`return` values allow us to do that

We can store the results of calculation through the use of `return` values

This is how methods like `.toUpperCase()` work

`return` Values

```
function squareNumber(num) {
  let square = num * num;
  return square;
}

let squareOfFour = squareNumber(4);

console.log(squareOfFour); // 16
```

`return` Values

Think of it like a conversation

In a conversation: we ask a question, and we sometimes get a response back

In JavaScript: we call a function, and we sometimes get a `return` value

`return` Values

```
function squareNumber(num) {
  let square = num * num;
  return square;
}

let squareOfFour = squareNumber(4);
let squareOfTwelve = squareNumber(12);

console.log(squareOfFour + squareOfTwelve); // 160

console.log(squareNumber(8) + squareNumber(11)); // 185
```

`return` Values

```
function cube(num) {
  return num * num * num;
}

function double(num) {
  return num * 2;
}

let result = double(cube(4)); // 128
```

`return` Values

```
let userOne = {
  admin: true,
};

let userTwo = {
  admin: false,
};

function isAdmin(user) {
  return user.admin === true;
}

isAdmin(userOne); // true
isAdmin(userTwo); // false
```

Passing in Variables

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

let firstNumber = 10;
let secondNumber = 24;

addTwoNumbers(firstNumber, secondNumber); // 34
```

Function Guidelines

Follow *F.I.R.S.T* principles:

- **F**ocused
- **I**ndependent
- **R**eusable
- **S**mall
- **T**estable

Also, try to make it fault-tolerant. But that doesn't fit in the acronym

Start the exercises here, please!

See you in 10 minutes

That's all for tonight!

Homework

- Conditionals
- Loops
- Arrays
- Objects
- Functions
- Read JavaScript.info's array page, array methods page, object page and function page
- Read Eloquent JavaScript's Object and Arrays chapter and the Functions chapter

Homework: Extra

- Read Eloquent JavaScript
- Read JavaScript.info
- Read Speaking JavaScript

What's Next?

- Pseudocode
- Advanced Functions
  - Callbacks
  - Scope and Hoisting
  - Closures
  - Higher Order Functions
  - Rest Parameters
  - Spread Operators

Thank You!