Introduction to Javascript

David Li

Vol 1

Introduction to Javascript

David Li

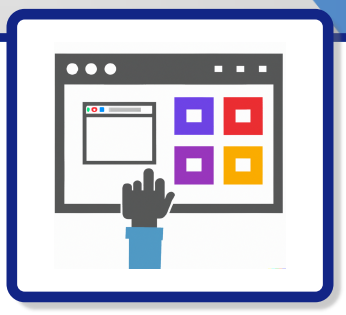
Vol 1

God made the integers; all else is the work of man.

Leopold Kronecker.

Contents

1. Introduction to Javascript



Any application that can be written in JavaScript will eventually be written in JavaScript. — Jeff Atwood

1.1 History of Javascript

JavaScript is a programming language that is commonly used to add interactive elements to websites. It is a client-side language, which means that it is executed by the user's web browser rather than on the server. JavaScript allows developers to create dynamic and interactive user experiences, such as changing the content of a web page without reloading the page, validating form input, and creating animations and games. It is commonly used in combination with HTML and CSS to create websites and web applications.

The full history of JavaScript is as follows:

1. 1995: JavaScript was developed by Netscape Communications Corporation as a programming language for web browsers. It was initially named LiveScript,

but was later renamed to JavaScript to capitalize on the popularity of the Java programming language.

2. 1996: Microsoft released its own version of JavaScript, called JScript, for its Internet Explorer web browser.
3. 1997: JavaScript was standardized by ECMA International, an industry organization for standardizing information and communication systems, as ECMAScript.
4. 1999: ECMAScript 3, the first widely-supported version of JavaScript, was released.
5. 2005: ECMAScript 5, which added many new features to JavaScript, was released.
6. 2009: ECMAScript 5.1, which was a minor update to ECMAScript 5, was released.
7. 2015: ECMAScript 6 (also known as ECMAScript 2015), which added many new features and improvements to JavaScript, was released.

Today, JavaScript continues to be a popular and widely-used programming language for web development.

1.1.1 Why use Javascript

There are several advantages of using JavaScript for full stack development:

1. Flexibility: JavaScript is a versatile language that can be used for both front-end and back-end development, allowing for a seamless development process and a cohesive codebase.

2. Popularity: JavaScript is one of the most popular programming languages, with a large and active community of developers who constantly contribute new tools and libraries to improve the language.

3. Ease of use: JavaScript is a relatively easy language to learn, even for those with little to no programming experience. This makes it a great choice for developers who want to learn how to build full stack applications.

4. Speed: JavaScript is a fast language that can execute code quickly, allowing for faster development and better performance in web applications.

5. Compatibility: JavaScript is supported by all major web browsers, so web applications built with JavaScript will be compatible with a wide range of devices and platforms.

6. Rich ecosystem: JavaScript has a rich ecosystem of frameworks, libraries, and tools that make it easier to build and maintain full stack applications. This includes popular frameworks such as React, Angular, and Vue.js for the front-end, and Node.js for the back-end.

JavaScript is a very popular and widely-used programming language, so knowing JavaScript is a valuable skill for a software developer to have. JavaScript is often used for building web applications and creating interactive user experiences on the front end (i.e., the client-side) of a web application.

In addition to being used for front-end web development, JavaScript is also commonly used for server-side development using runtime environments like Node.js. This allows JavaScript to be used for full-stack development, which can be beneficial for developers who want to be able to work on both the front-end and back-end of a web application.

Overall, knowing JavaScript can be very useful for a software developer, and it is a skill that is in high demand in the job market. However, the importance of any particular language or technology can vary depending on the specific job and industry, so it's always a good idea to keep up with the latest trends and developments in the field.

1.1.2 Getting started with Javascript

JavaScript is a client-side language, which means that it is executed by the user's web browser rather than on the server. Therefore, there is no need to install JavaScript on your computer.

To use JavaScript in a web page, you simply need to include the JavaScript code in the HTML code of the page. This can be done by adding a `<script>` tag in the `<head>` or `<body>` section of the HTML code, and then placing the JavaScript code inside the `<script>` tag. For example:

```
<script>
    // JavaScript code goes here
</script>
```

Listing 1.1: sample script tag

Alternatively, you can also include a reference to a separate JavaScript file in the <head> or <body> section of the HTML code using the src attribute of the <script> tag. For example:

```
<script src="script.js"></script>
```

This will include the JavaScript code from the script.js file in the web page.

To test your JavaScript code, you can simply open the web page in a web browser and use the browser's developer tools to view the output of the code. Most modern

web browsers, such as Google Chrome and Mozilla Firefox, have developer tools built-in that allow you to view and debug your JavaScript code.

What is HTML

1.1.3 Getting started with Node.js

is an open-source, cross-platform, runtime environment that allows you to execute JavaScript code outside of a browser. It provides a rich set of built-in modules that simplify the development of web applications, and it has a large and active community of users who contribute additional modules to the npm (Node Package Manager) repository.

is often used to build server-side applications, but it can also be used for command-line tools, desktop applications, and more. It is based on the JavaScript V8 engine, which is the same engine used in the Google Chrome web browser, and it allows you to write code in JavaScript that can access the full range of system-level APIs and libraries.

One of the key advantages of Node.js is its event-driven, non-blocking I/O model, which makes it highly scalable and efficient. This makes it well-suited for building real-time, data-intensive applications that can handle large numbers of concurrent connections [3].

Overall, is a powerful and versatile tool that can be used for a wide range of applications. For details on how to install node view chapter 3 on page 53

1.2 Javascript Fundamentals

There are several fundamental concepts in JavaScript that are important for a beginner to understand. These include:

1. Variables: Variables are containers for storing data values. In JavaScript, you use the `var` keyword to declare a variable and assign it a value using the = operator. For example:

```
var x = 5;
```

2. Data types: JavaScript has several data types that can be used to represent different types of data. These include numbers, strings (text), booleans (true/false values), arrays, and objects. The type of a variable is determined by the data it holds.

3. Operators: Operators are used to perform operations on data values. For example, the + operator is used to add two numbers together, while the === operator is used to check if two values are equal.

4. Functions: Functions are blocks of code that can be called from other parts of your program. In JavaScript, you define a function using the function keyword, followed by the function name and a set of parentheses that may contain parameters. For example:

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

Listing 1.2: sample function

5. Control flow: Control flow refers to the order in which the statements in a program are executed. JavaScript uses conditional statements (e.g., if, else, switch) and loops (e.g., for, while) to control the flow of a program.

6. Objects: As mentioned earlier, objects are collections of properties that have values of various data types. In JavaScript, you can create your own objects and add, remove, or modify their properties.

In JavaScript, , , and are three different ways to declare variables.

var is the traditional way to declare variables in JavaScript. It has been around since the early days of the language and is still used in many programs. The main disadvantage of using var is that it is function-scoped, which means that a variable declared with var inside a function is accessible outside of that function. This can lead to unexpected behavior and can make it difficult to understand and debug your code.

 is a newer way to declare variables in JavaScript. It was introduced in the ES6 (ECMAScript 6) version of the language and is now the recommended way to declare variables in most cases. let is block-scoped, which means that a variable declared with let inside a block of code (e.g. inside a for loop or an if statement) is only accessible within that block. This makes let more predictable and easier to understand than var.

 is also a new way to declare variables in JavaScript. It was also introduced in ES6 and is similar to let, but with one key difference: a variable declared with const cannot be reassigned. In other words, once you assign a value to a const variable, you cannot change that value later on. This makes const a good choice for variables that you don't want to change, such as constant values or configuration settings.

In summary, var is the traditional way to declare variables in JavaScript, but it has

some limitations and can lead to unpredictable behavior. let and const are newer ways to declare variables that are more predictable and easier to understand. In most cases, it is recommended to use let unless you have a specific reason to use var or .

These are just a few of the fundamental concepts in JavaScript. To learn more, I would recommend reading some tutorials available at mdn [1].

1.2.1 Javascript Objects

In JavaScript, an object is a collection of properties, and a property is an association between a name (or key) and a value. A property's value can be a function, in which case the property is known as a method. In addition to objects that are predefined in the browser, you can define your own objects.

Here is an example of a simple object:

```
var person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};
```

In this example, person is an object that has four properties: firstName, lastName, age, and eyeColor. The values of these properties are "John", "Doe", 50, and "blue", respectively.

You can access an object's properties in two ways: using dot notation (e.g., person.firstName) or bracket notation (e.g., person["firstName"]). In general, dot notation is preferred, but if the property name contains spaces or other special characters, you must use bracket notation.

To use JavaScript objects, you can declare a variable and set it equal to a new object using curly braces. For example:

```
var myObject = {};
```

To add properties to the object, you can use the dot notation or the bracket notation. For example:

```
// Dot notation
myObject.name = "John Doe";
myObject.age = 30;

// Bracket notation
myObject["gender"] = "male";
```

Listing 1.3: Dot notiation in javascript

To access the properties of an object, you can use the same dot notation or bracket notation. For example:

```
console.log(myObject.name); // Output: "John Doe"
console.log(myObject["age"]); // Output: 30
```

You can also use the for...in loop to iterate over the properties of an object and access their values. For example:

```
for (var key in myObject) {
console.log(key + ": " + myObject[key]);
}

// Output:
// name: John Doe
// age: 30
// gender: male
```

Additionally, you can use the Object.keys() method to get an array of the keys in an object, and the Object.values() method to get an array of the values in an object. For example:

```
1 console.log(Object.keys(myObject)); // Output: ["name", "age", "
    gender"]
2 console.log(Object.values(myObject)); // Output: ["John Doe", 30, "
    male"]
```

JavaScript objects are a useful data structure for storing and organizing data, and are commonly used in web development.

1.2.2 Arrays in Javascript

What is an array? An array is a data structure that allows you to store a collection of elements in a single variable. This can be useful in many situations, such as when you want to store a list of items or when you need to store data in a structured manner.

Here are some reasons why you might want to use an array in your code:

To store a list of items: An array is a convenient way to store a list of similar items. For example, you could use an array to store a list of names, a list of numbers, or a list of other objects.

To access items by index: Arrays are indexed, which means that you can access each element in the array using a numerical index. This makes it easy to retrieve specific items from the array, or to loop through all of the items in the array.

To sort items: Arrays have built-in methods for sorting the items they contain. This can be useful if you need to sort a list of items alphabetically or numerically.

To manipulate items: Arrays also have methods for adding, removing, and modifying the items they contain. This makes it easy to manipulate the items in the array

without having to write your own code to do so.

Overall, arrays are a versatile data structure that can be useful in many situations where you need to store and manipulate data. If you find yourself needing to store and manage a collection of items, an array may be the right choice for your needs.

Properties of arrays in Javascript One of the unique aspects of JavaScript arrays is that they can store elements of different data types. In many programming languages, arrays are limited to storing elements of a single data type, but in JavaScript, an array can contain elements of any type.

For example, the following code creates an array that contains a number, a string, and a boolean value:

```
1 var array=[4, "test", false]
```

One key difference between arrays in JavaScript and other programming languages is that JavaScript arrays are dynamic, which means that they can grow or shrink in size as needed. This is in contrast to arrays in languages like C or Java, which have a fixed size and must be explicitly resized if the number of elements exceeds the size of the array.

Below is an example highlighting what you can do with arrays in javascript.

Some of the most commonly used functions of arrays in JavaScript include the following:

- push(): Adds one or more elements to the end of an array.
- pop(): Removes the last element from an array and returns it.
- shift(): Removes the first element from an array and returns it.
- unshift(): Adds one or more elements to the beginning of an array.

- indexOf(): Returns the index of the first occurrence of a given element in an array, or -1 if the element is not present in the array.
- join(): Joins all elements of an array into a string and returns the resulting string.
- slice(): Extracts a portion of an array and returns a new array.
- splice(): Removes elements from an array and/or adds new elements to an array.
- sort(): Sorts the elements of an array in ascending or descending order.

These are just a few examples of the many functions available for working with arrays in JavaScript. Other commonly used array functions include map(), filter(), reduce(), forEach(), and many more. The specific functions you use will depend on your specific needs and the tasks you are trying to accomplish with your arrays.

```
Here is an example of using an array in javascript:

// Declare an array with 3 elements
var myArray = [1, 2, 3];

// Output the first element in the array
console.log(myArray[0]); // Output: 1

// Add a new element to the end of the array
myArray.push(4);

// Output the length of the array
console.log(myArray.length); // Output: 4

```

```
15 // Use the slice() method to create a new array with the last two
       elements of the original array
16 var lastTwo = myArray.slice(myArray.length-2);
17
18 // Output the new array
19 console.log(lastTwo); // Output: [3, 4]
20
21 // Use the map() method to create a new array with the square of
       each element in the original array
22 var squares = myArray.map(x => x*x);
23
24 // Output the new array
25 console.log(squares); // Output: [1, 4, 9, 16]
26
27 // Use the filter() method to create a new array with only the even
       elements in the original array
28 var evens = myArray.filter(x => x % 2 === 0);
29
30 // Output the new array
31 console.log(evens); // Output: [2, 4]
32
33 // Use the reduce() method to sum all of the elements in the
       original array
34 var sum = myArray.reduce((total, current) => total + current);
35
36 // Output the sum
37 console.log(sum); // Output: 10
38
39 // Use the sort() method to sort the elements in the original array
       in ascending order
40 myArray.sort((a, b) => a - b);
```

```
41
42 // Output the sorted array
43 console.log(myArray); // Output: [1, 2, 3, 4]
44
45 // Use the reverse() method to reverse the order of the elements in
       the original array
46 myArray.reverse();
47
48 // Output the reversed array
49 console.log(myArray); // Output: [4, 3, 2, 1]
```

Listing 1.4: Javascript array example

Here is an example of using the reduce() function to reduce an array of objects in JavaScript:

```
1  const data = [
2    {
3      name: "John Doe",
4      age: 34,
5      city: "New York"
6    },
7    {
8      name: "Jane Smith",
9      age: 29,
10     city: "Los Angeles"
11   },
12   {
13     name: "Bob Johnson",
14     age: 42,
15     city: "Chicago"
16   }
17 ];
```

```
18
19 const totalAge = data.reduce((total, person) => {
20   return total + person.age;
21 }, 0);
22
23 console.log(totalAge);  // Output: 105
```

Listing 1.5: Example of reducing an array in javascript

In this example, we have an array of objects representing people, with each object containing a name, age, and city. We use the reduce() function to iterate over the array of objects and calculate the total age of all the people. The reduce() function takes a callback function and an initial value (in this case, 0) as arguments, and applies the callback function to each element in the array. In this case, the callback function adds the age property of each person object to the total variable, which is initially set to 0. The final result of the reduce() function is the total age of all the people in the array.

This is just one example of how the reduce() function can be used to reduce an array of objects in JavaScript. The specific implementation will depend on the data you are working with and the calculation you want to perform.

1.2.3 Control Flow in Javascript

Control flow is a fundamental concept in programming that involves specifying the order in which different parts of a program are executed. In JavaScript, control flow is typically implemented using control flow statements such as if...else, switch, for, and while loops.

Control flow is important because it allows you to create programs that can make decisions and execute different code based on certain conditions. This is essen-

tial for creating programs that can adapt to different inputs and scenarios, and for performing complex tasks that require multiple steps or iterations.

Here are a few examples of why you might use control flow in JavaScript:

1. To check the value of a variable and execute different code depending on its value. For example, you might use an if...else statement to check the value of a user's input and respond differently depending on whether the input is valid or not.

2. To execute the same code multiple times with different values. For example, you might use a for loop to iterate over a list of items and perform the same operation on each item in the list.

3. To execute code repeatedly until a certain condition is met. For example, you might use a while loop to keep checking the value of a variable until it reaches a certain threshold, at which point the loop will stop.

Here is an example of using a switch statement in JavaScript:

```
const day = "Saturday";

switch (day) {
  case "Monday":
    console.log("Today is Monday");
    break;
  case "Tuesday":
    console.log("Today is Tuesday");
    break;
  case "Wednesday":
    console.log("Today is Wednesday");
    break;
```

```
13    case "Thursday":
14      console.log("Today is Thursday");
15      break;
16    case "Friday":
17      console.log("Today is Friday");
18      break;
19    case "Saturday":
20      console.log("Today is Saturday");
21      break;
22    case "Sunday":
23      console.log("Today is Sunday");
24      break;
25    default:
26      console.log("Invalid day");
27  }
28
29  // Output: Today is Saturday
```

Listing 1.6: Example switch statement in Javascript

In this example, we have a variable day that contains the current day of the week. We use a switch statement to check the value of day and execute different code depending on the day of the week. The switch statement takes a value as its input and compares it to the case labels inside the switch block. If the value matches one of the case labels, the code associated with that case label is executed. If the value does not match any of the case labels, the code associated with the default label is executed (if it is present).

In this example, the value of day is "Saturday", so the code inside the case "Saturday" block is executed and the message "Today is Saturday" is logged to the console. The break statement at the end of each case block is used to prevent the code from "falling

through" to the next case block. If the break statement was not present, the code inside all the case blocks after the matching case block would also be executed.

This is just one example of how a switch statement can be used in JavaScript. The specific implementation will depend on your specific needs and the values you are working with.

Here is an example of using an if...else statement in JavaScript:

```
const x = 5;

if (x > 10) {
  console.log("x is greater than 10");
} else {
  console.log("x is less than or equal to 10");
}

// Output: x is less than or equal to 10
```

Listing 1.7: Example if else statement

In this example, we have a variable x with the value 5. We use an if...else statement to check the value of x and execute different code depending on whether x is greater than 10 or not. The if part of the if...else statement specifies a condition (in this case, $x > 10$) that is evaluated to either true or false. If the condition is true, the code inside the if block is executed. If the condition is false, the code inside the else block is executed.

In this example, the value of x is 5, which is less than or equal to 10, so the condition $x > 10$ is false and the code inside the else block is executed. This logs the message "x is less than or equal to 10" to the console.

This is just one example of how an if...else statement can be used in JavaScript. The

specific implementation will depend on your specific needs and the values you are working with.

Here is an example of using a for loop in JavaScript:

```
const numbers = [1, 2, 3, 4, 5];

for (let i = 0; i < numbers.length; i++) {
  console.log(numbers[i]);
}

// Output:
// 1
// 2
// 3
// 4
// 5
```

Listing 1.8: Example of for loop in Javascript

In this example, we have an array of numbers called numbers. We use a for loop to iterate over the array and print each number to the console. The for loop takes three parts: an initialization (let i = 0), a condition (i < numbers.length), and an update (i++). The initialization part is executed once when the loop starts, and sets the initial value of the loop variable (i in this case). The condition is evaluated before each iteration of the loop, and the loop continues as long as the condition is true. The update is executed after each iteration of the loop, and is used to update the value of the loop variable.

In this example, the initialization part sets the initial value of i to 0. The condition part checks if i is less than the length of the numbers array (which is 5 in this case). As long as i is less than 5, the loop will continue to run. The update part increments

the value of i by 1 after each iteration of the loop.

Inside the loop body, we use the current value of i as the index of the numbers array to access and log the corresponding element to the console. In the first iteration, i is 0, so the element at index 0 (which is 1) is logged to the console. In the second iteration, i is 1, so the element at index 1 (which is 2) is logged to the console, and so on.

This is just one example of how a for loop can be used in JavaScript. The specific implementation will depend on your specific needs and the data you are working with.

Here is an example of using a while loop in JavaScript:

```
let x = 5;

while (x > 0) {
  console.log(x);
  x--;
}

// Output:
// 5
// 4
// 3
// 2
// 1
```

Listing 1.9: Example of while loop in Javascript

In this example, we have a variable x with the initial value 5. We use a while loop to iterate as long as x is greater than 0, and print the value of x to the console on each iteration. The while loop takes a condition as its input (in this case, $x > 0$),

and continues to run as long as the condition is true.

Inside the loop body, we first log the current value of x to the console using console.log(). Then we use the x– statement to decrement the value of x by 1. This is important because the value of x must change in some way on each iteration of the loop, or the loop will run indefinitely (i.e. it will create an infinite loop).

In this example, the loop starts with x equal to 5. The condition $x > 0$ is true, so the code inside the loop body is executed and the value of x (5) is logged to the console. Then the value of x is decremented by 1, so x is now equal to 4. The condition $x > 0$ is still true, so the code inside the loop body is executed again and the new value of x (4) is logged to the console. This process continues until the value of x is 0, at which point the condition $x > 0$ is false and the loop stops.

This is just one example of how a while loop can be used in JavaScript. The specific implementation will depend on your specific needs and the data you are working with.

Overall, control flow is a crucial aspect of programming in JavaScript and other languages. It allows you to create programs that are more flexible, adaptable, and powerful, and is an essential tool for solving complex problems and performing complex tasks.

1.2.4 Truthy and Falsy values

In JavaScript, a boolean value is a value that is either true or false. You can convert a value of any other type to a boolean value using the Boolean() function.

Here is an example of how to use the Boolean() function to convert a value to a boolean:

```
1 var myValue = 'hello';
```

```
var myBoolean = Boolean(myValue);
```

In this example, the string value "hello" is converted to a boolean value using the Boolean() function. The resulting boolean value will be true, because the string "hello" is a non-empty value.

You can also use the logical operators !, &&, and || to convert a value to a boolean. For example, the ! operator will convert a value to false if it is true, and true if it is false. The && and || operators will convert a value to true if it is truthy (i.e. if it evaluates to true when used in a boolean context), and false if it is falsy (i.e. if it evaluates to false when used in a boolean context).

Here are some examples of using the !, &&, and || operators to convert values to booleans:

```
var myValue = 'hello';
var myBoolean1 = !myValue;  // myBoolean1 is false
var myBoolean2 = myValue && true;  // myBoolean2 is true
var myBoolean3 = myValue || false;  // myBoolean3 is true
```

Listing 1.10: Boolean validation

In these examples, the ! operator converts the value of myValue to false because it is non-empty. The && operator converts the value of myValue to true because it is truthy. And the || operator also converts the value of myValue to true because it is truthy.

In JavaScript, a truthy value is a value that is considered to be true when used in a boolean context. This means that if you use a truthy value in a conditional statement, such as an if statement, the condition will evaluate to true.

Conversely, a falsy value is a value that is considered to be false when used in a boolean context. If you use a falsy value in a conditional statement, the condition

will evaluate to false.

Here are some examples of truthy and falsy values in JavaScript:

- true is a truthy value.
- false is a falsy value.
- The number 0 is a falsy value.
- The empty string '' is a falsy value.
- The null value is a falsy value.
- The undefined value is a falsy value.

In JavaScript, there are six falsy values: false, 0, '', null, undefined, and NaN (not a number). All other values are truthy, including non-empty strings, numbers, and objects [2].

You can use the Boolean() function to convert a value to a boolean and determine whether it is truthy or falsy. For example:

```
1 var myValue = 'hello';
2 var myBoolean = Boolean(myValue);  // myBoolean is true
```

In this example, the string 'hello' is a truthy value, so the Boolean() function returns true when called on myValue. You can also use the logical operators !, &&, and || to convert a value to a boolean and determine whether it is truthy or falsy. For example:

1.3 HTML and CSS

HTML, which stands for Hypertext Markup Language, is a language used to create the structure and content of a web page. It is the standard markup language for

creating web pages and web applications.

HTML consists of a series of elements that are used to define the different parts of a web page, such as the text, images, headings, links, and other content. These elements are represented by tags, which are enclosed in angle brackets and typically come in pairs, with an opening and closing tag.

For example, the <h1> tag is used to create a level 1 heading, and the <p> tag is used to create a paragraph. The following HTML code creates a page with a heading and a paragraph:

```
1 <h1>My page</h1>
2 <p>This is my page.</p>
```

Listing 1.11: Heading and Paragraph tag

HTML is typically used in conjunction with other languages, such as CSS (Cascading Style Sheets) and JavaScript, to create the complete user interface of a web page. These languages are used to define the look and behavior of the page, respectively. HTML is a fundamental technology of the World Wide Web, and is supported by all modern web browsers. It is used to create and structure the content of millions of web pages on the internet.

There are many different HTML elements that are commonly used to create the structure and content of a web page. Some of the most common HTML elements are:

- <h1> - <h6>: These elements are used to create headings of different levels. The <h1> element is the main heading of a page, and the <h6> element is the lowest level heading.
- <p>: This element is used to create a paragraph of text.

- <a>: This element is used to create a hyperlink to another web page or to a specific location on the current page.
- <img>: This element is used to embed an image on a web page.
- : This element is used to create a container for other HTML elements, and is often used to group elements together and apply styles to them.
- <form>: This element is used to create a form that allows users to enter data, which can then be submitted to a server for processing.
- <input>: This element is used to create various types of input fields, such as text boxes, checkboxes, and radio buttons, within a <form> element.
- <button>: This element is used to create a button that can be clicked by the user to perform an action, such as submitting a form or triggering a JavaScript function.
- <table>: This element is used to create a table to display data in a grid of rows and columns.
- : This element is used to create an unordered list, which is a list of items that are not presented in a specific order.
- : This element is used to create an ordered list, which is a list of items that are presented in a specific order, such as a numbered list.
- : This element is used to create a list item within a <ul> or <ol> element. These are just some of the many common HTML elements that are used to create the structure and content of a web page. There are many other

elements that are used for specific purposes, such as creating video or audio players, or for adding semantic meaning to the page.

1.3.1 CSS

CSS, or Cascading Style Sheets, is a stylesheet language used for describing the look and formatting of a document written in a markup language. It is most commonly used to style web pages written in HTML and XHTML, but can also be used with other markup languages like SVG.

CSS allows you to define the styles for elements in a document, such as the colors, fonts, and layout, and apply those styles consistently across multiple pages or elements. This makes it possible to separate the content of a document from its formatting, allowing you to easily change the look and feel of a website without having to modify the content of the pages.

CSS uses a set of rules, called "selectors," to apply styles to specific elements in a document. These rules specify which elements the styles should be applied to, and can be based on the element's type, id, class, or other attributes.

For example, a CSS rule might look like this:

```
1 h1 {
2   color: blue;
3   font-size: 24px;
4 }
```

In this example, the h1 selector is used to apply the color and font-size styles to all <h1> elements in the document. The styles are specified as a list of properties and values, separated by colons, and are enclosed in curly braces.

CSS is a powerful and versatile tool for styling and formatting documents written in markup languages. It allows you to create consistent and attractive designs for your

websites, and to easily change the look and feel of those designs without modifying the content of the pages.

In CSS, a class is a group of elements that are defined with the same style attributes. These styles can then be applied to any element on a page by simply applying the class to that element. This allows you to define the styles once and then use them multiple times on different elements, rather than having to define the styles for each element individually.

On the other hand, the global scope in CSS refers to the styles that are applied to elements on a page by default. These styles are applied to all elements on the page unless they are specifically overruled by other styles, such as those defined in a class. The global scope in CSS is determined by the styles that are included in the default stylesheet for the page, as well as any additional stylesheets that are included in the page.

In summary, CSS classes allow you to define styles that can be applied to multiple elements on a page, while the global scope in CSS refers to the default styles that are applied to all elements on a page.

Inline CSS is a method of applying styles to an HTML element by using the style attribute in the HTML markup. This attribute takes a string of CSS rules that are applied directly to the element, rather than being defined in a separate stylesheet. Here is an example of inline CSS:

```
1 <p style="color: red; font-size: 16px;">
2   This paragraph will be displayed in red and with a font size of 16
      px.
3 </p>
```

Listing 1.12: Inline CSS

In this example, the style attribute is used to apply the color and font-size styles directly to the <p> element. The styles are specified as a string of CSS rules, separated by semicolons.

Inline CSS has some advantages over other methods of applying styles to HTML elements. For example, it allows you to apply styles to a single element without having to define them in a separate stylesheet, which can be useful when you only need to apply a small number of styles to a specific element.

However, inline CSS also has some disadvantages. Because the styles are applied directly to the elements in the HTML markup, it can make the markup more difficult to read and maintain. Additionally, using inline CSS means that you have to repeat the same styles on multiple elements if you want them to have the same styling, which can be tedious and makes it difficult to update the styles consistently across the page.

Overall, it is generally considered better practice to define styles in a separate stylesheet, rather than using inline CSS, unless you have a specific reason to do so.

To include a CSS file in an HTML page, you can use the link element within the head section of your HTML page. The link element should have the rel attribute set to "stylesheet" and the href attribute set to the URL of the CSS file.

```
<head>
  <link rel="stylesheet" href="styles.css">
</head>
```

Listing 1.13: CSS stylesheet in html

In this example, the <link> element is used to link to a separate stylesheet file called styles.css. This file contains the CSS rules that will be applied to the elements in the HTML document.

Another way to include CSS in an HTML document is to use the <style> element in the <head> of the document, like this:

```
<head>
  <style>
    /* CSS rules go here */
  </style>
</head>
```

Listing 1.14: CSS styles included html

In this example, the <style> element is used to define the CSS rules directly in the HTML document, rather than in a separate stylesheet file. This can be useful when you only have a small amount of CSS to include, or when you want to apply styles that are only used in a specific page or section of the document.

Overall, the best method for including CSS in an HTML document depends on the specific needs of your project. Using a separate stylesheet allows you to keep your CSS rules in a separate file, which can make them easier to maintain and update. However, using the <style> element can be useful when you only have a small amount of CSS to include, or when you want to apply styles that are specific to a single page or section of the document.

Once you have included your CSS file in your HTML page, the styles will be applied to the page when it is loaded in a web browser. You can then modify the styles in the CSS file and refresh the page to see the changes.

```
p {
  font-family: Arial;
  color: #333;
}
```

Listing 1.15: Heading and Paragraph tag

In this example, the p selector selects all p elements on the page, and the declaration applies the font-family and color styles to the selected elements. The font-family style sets the font of the text to Arial, and the color style sets the color of the text to a dark gray.

Here is an example that demonstrates the difference between CSS classes and the global scope in CSS. Let's say you have a page with the following elements:

```
<h1>Heading 1</h1>
<p>Paragraph 1</p>
<p class="highlight">Paragraph 2</p>
<p>Paragraph 3</p>
```

In this example, the <h1> and <p> elements are not assigned to any CSS classes, so they will be styled according to the global scope. Let's say the global scope defines the following styles:

```
h1 {
  color: blue;
  font-size: 24px;
}

p {
  color: black;
  font-size: 16px;
}
```

In this case, the <h1> element will be displayed in blue and with a font size of 24px, and the <p> elements will be displayed in black and with a font size of 16px. Now, let's say you have defined a CSS class called highlight with the following styles:

```
.highlight {
```

```
  color: red;
  font-weight: bold;
}
```

You can apply this class to the <p> element with the class="highlight" attribute, like this:

```
<p class="highlight">Paragraph 2</p>
```

In this case, the <p> element with the highlight class will be displayed in red and with a bold font, overriding the styles defined in the global scope. The other <p> elements on the page will still be styled according to the global scope, so they will be displayed in black and with a font size of 16px.

So, in this example, the CSS class highlight allows you to define a specific set of styles that can be applied to one or more elements on the page, while the global scope defines the default styles that are applied to all elements on the page unless they are overruled by other styles.

CSS is a fundamental technology of the World Wide Web, and is supported by all modern web browsers. It is used to create the visual appearance of millions of web pages on the internet.

1.4 Mixing Javascript with HTML

Here is an example of an HTML page with a button that pops out an alert when clicked: CSS, and JavaScript:

```
<html>
  <head>
    <title>Button Alert</title>
  </head>
```

```
  <body>
    <button id="alert-button">Click me</button>
    <script>
      const alertButton = document.getElementById('alert-button');
      alertButton.addEventListener('click', () => {
        alert('Button clicked!');
      });
    </script>
  </body>
</html>
```

Listing 1.16: Alert with HTML

In this example, the HTML page contains a button element with the ID alert-button. When the user clicks the button, the addEventListener() method is used to register a click event listener that pops up an alert using the alert() function.

When the user clicks the "Click me" button, the alert() function is called and a popup window appears with the message "Button clicked!". The user can then click the "OK" button to close the alert and continue using the page.

1.5 Logging in Javascript

In JavaScript, advanced logging typically refers to using more advanced features of the console object to log information to the console. This can include using different logging methods (such as console.warn() and console.error()), using formatting options (such as console.table()), and using console methods to group and organize log messages (such as console.group() and console.groupEnd()).

Here are a few examples of advanced logging in JavaScript:

- Using different logging methods: The console object in JavaScript has multiple

methods for logging different types of messages to the console. For example, you can use console.log() to log regular messages, console.warn() to log warning messages, and console.error() to log error messages. These methods have different visual styles in the console (e.g. warning messages are yellow and error messages are red), which can make it easier to quickly identify different types of messages.

```
1 console.log("Regular message");
2 console.warn("Warning message");
3 console.error("Error message");
```

Listing 1.17: Example of console.log

- Using formatting options: The console object also has methods for formatting and displaying data in different ways. For example, you can use console.table() to log an array of objects as a table, console.dir() to display the properties of an object, and console.time() and console.timeEnd() to measure the time it takes to run a piece of code. These methods can make it easier to read and understand complex data, and can be useful for debugging and performance analysis.

```
1  const data = [
2    {
3      name: "John Doe",
4      age: 34,
5      city: "New York"
6    },
7    {
8      name: "Jane Smith",
9      age: 29,
10     city: "Los Angeles"
```

```
  },
  {
    name: "Bob Johnson",
    age: 42,
    city: "Chicago"
  }
];

console.table(data);
console.dir(data[0]);

console.time("Time");
// Code to measure goes here
console.timeEnd("Time");
```

Listing 1.18: Example of console table and dir

- Using grouping methods: The console object also has methods for grouping and organizing log messages. For example, you can use console.group() and console.groupEnd() to create a nested group of log messages that can be expanded or collapsed in the console. This can make it easier to organize and read large numbers of log messages, and can be useful for isolating specific sections of your code for debugging or analysis.

```
console.group("Group 1");
console.log("Message 1");
console.log("Message 2");

console.group("Group 2");
console.log("Message 3");
console.log("Message 4");
```

```
8  console.groupEnd();
9
10 console.log("Message 5");
11 console.groupEnd();
```

Listing 1.19: Usage of console.group

In this example, we create two groups of log messages using console.group() and console.groupEnd(). The first group has the label "Group 1" and contains two log messages ("Message 1" and "Message 2"). The second group has the label "Group 2" and contains two log messages ("Message 3" and "Message 4"). The second group is nested inside the first group, so it is displayed as a sub-group in the console. The final log message ("Message 5") is not part of any group, so it is displayed at the same level as the first group in the console.

When you run this code and view the browser console (Chrome), you should see the following output:

```
Group 1
    Message 1
    Message 2
    Group 2
        Message 3
        Message 4
    Message 5
```

You can click on the group label (e.g. "Group 1") in the console to expand or collapse the group and show or hide the log messages inside the group.

This can make it easier to read and organize large numbers of log messages, and can be useful for isolating specific sections of your code for debugging or analysis.

Overall using console.log() in JavaScript is important for several reasons:

- It allows you to see the output of your code as it is executed, which can be useful for debugging and understanding how your code is working.
- It allows you to inspect and interact with the data in your code, which can be useful for testing and verifying that your code is working as expected.
- It allows you to log messages, warnings, and errors to the console, which can be useful for tracking the progress and status of your code.
- It allows you to measure the performance of your code, which can be useful for identifying and optimizing bottlenecks and inefficiencies.
- In general, using console.log() (and the other console methods) is an essential tool for any JavaScript developer. It provides valuable information and feedback that can help you write better, more reliable, and more efficient code.

To test your knowledge please answer the following questions

Question 1 Find the number that are greater than 10

```
1 const numbers = [5, 10, 15, 20, 25];
2 // find value of greaterThan10
3 // const greaterThan10 = []
4 console.log(greaterThan10); // [15, 20, 25]
```

See solution 1 at page 66.

Question 2 What is the output of the following code

```
1 console.group('My Group');
2 console.log('This is the first log in my group');
3 console.log('This is the second log in my group');
4 console.groupEnd();
```

You should be able to view the output in your browser console.

See solution 1 at page 67.

2. Asynchronous Programming

> The only way to learn a new programming language is by writing programs in it
>
> — Dennis Ritchie

Asynchronous programming in JavaScript refers to the concept of non-blocking I/O operations. This means that when an asynchronous operation is performed, the program continues to execute the next instruction without waiting for the asynchronous operation to complete. This can be achieved using callbacks, promises, and async/await.

2.1 Introduction to Promises

A JavaScript promise is an object that represents the eventual result of an asynchronous operation. A promise can be in one of three states: fulfilled, rejected, or pending. A fulfilled promise means that the asynchronous operation has completed successfully and a value is available. A rejected promise means that the asynchronous operation has failed and an error is available. A pending promise means that the asynchronous operation is still in progress.

Promises are a better alternative to callback functions for handling asynchronous operations in JavaScript, because they make it easier to write and maintain code that uses asynchronous operations. Promises provide a cleaner and more intuitive syntax for working with asynchronous operations, and they can be composed together to create complex asynchronous behavior.

Why use promises Callback hell is a term used to describe the problem of deeply nested callback functions in JavaScript code. This can make the code difficult to read and maintain, and can lead to problems with the execution order of the asynchronous operations. To avoid callback hell, it is recommended to use the async/await syntax introduced in ES2017, or to use promises and the Promise.then() method.

Here is an example of using promises in JavaScript:

```
const myPromise = new Promise((resolve, reject) => {
  // do something asynchronous
  if (/* asynchronous operation was successful */) {
    resolve(/* result of the asynchronous operation */);
  } else {
    reject(/* error occurred during the asynchronous operation */);
  }
});

myPromise
  .then((result) => {
    // do something with the result of the promise
  })
  .catch((error) => {
    // handle any error that occurred during the promise
```

```
16 });
```

Listing 2.1: Example of promises

In the example above, the myPromise object is created with a function that performs an asynchronous operation. The function takes two arguments, resolve and reject, which are used to signal the completion or failure of the asynchronous operation. The then() method is used to specify a callback function that is called when the promise is fulfilled (i.e., the asynchronous operation is successful), and the catch() method is used to specify a callback function that is called if the promise is rejected (i.e., an error occurred during the asynchronous operation).

// usage of promise

2.2 Http Requests

HTTP, or Hypertext Transfer Protocol, is a networking protocol that is used to transfer data on the web. HTTP requests are messages sent by a client, such as a web browser, to a server to request information or perform actions. The server then responds to the request with an HTTP response message.

There are several different types of HTTP requests, each of which is used for a specific purpose. The most common types of HTTP requests are:

- GET: A GET request is used to retrieve data from a server. This type of request is typically used to retrieve a web page or other resource from a server.
- POST: A POST request is used to send data to a server for processing. This type of request is typically used when a user submits a form on a web page, and the data from the form is sent to the server for processing.

- PUT: A PUT request is used to update a resource on a server. This type of request is typically used to update an existing web page or other resource on a server.

- DELETE: A DELETE request is used to delete a resource on a server. This type of request is typically used to remove a web page or other resource from a server.

These are the most common types of HTTP requests, but there are many other types of requests that can be used for different purposes. HTTP requests are an important part of how the web works, as they allow clients and servers to communicate and exchange information.

What is an Endpoint An endpoint is a specific URL that is used to access a web service or API. An endpoint typically specifies the location of a specific resource or service on a server, and includes any necessary parameters or query string values. For example, consider a web service that allows users to search for books by title. The endpoint for this service might be something like https://example.com/books?title=harry+potter, where https://example.com/books is the base URL for the service, and title=harry+potter is a query string parameter that specifies the search term.
In this example, the endpoint is the full URL that is used to access the book search service. When a client, such as a web browser, makes an HTTP request to this endpoint, the server responds with the search results for the specified query.
Endpoints are an important part of how web services and APIs work, as they provide a way for clients to access the specific resources or services that are offered by the server. Endpoints typically include the base URL for the service, as well as any necessary parameters or query string values, to specify the exact resource or action

that is being requested.

To make an HTTP request in , you can use the object or the fetch API. Here's an example of how to use to make a GET request to fetch some data from a server:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'https://www.example.com/api/data', true);

xhr.onload = function() {
  if (this.status == 200) {
    var data = JSON.parse(this.response);
    // do something with the data
  }
};

xhr.send();
```

Listing 2.2: How to make a HTTP request in Javascript

Here's an example of how to use the fetch API to make the same request:

```
fetch('https://www.example.com/api/data')
  .then(response => response.json())
  .then(data => {
    // do something with the data
  });
```

Listing 2.3: HTTP request using fetch

Both and fetch allow you to specify additional options such as the request headers, and you can use them to make other types of HTTP requests such as POST, PUT, and DELETE.

2.2.1 Using Fetch

In JavaScript, the fetch() method is used to perform HTTP requests. It is a modern way to make network requests to retrieve resources from a server. fetch() is similar to other web request APIs like XMLHttpRequest (XHR).

The PokeAPI is a free and open-source API for accessing data about the Pokémon video game series. The API provides a GraphQL endpoint that allows you to query the API using the GraphQL language.

Here is an example of using the fetch() method to retrieve data about a Pokemon from the PokeAPI:

```
1 fetch('https://pokeapi.co/api/v2/pokemon/1')
2   .then(response => response.json())
3   .then(data => {
4     console.log(data);
5     // do something with the data here
6   });
```

Listing 2.4: Fetching entry from poke api

In this example, the fetch() method is used to make a GET request to the PokeAPI to retrieve information about the Pokemon with the ID of 1, which is Bulbasaur. The response.json() method is used to parse the response as JSON, and then the data is logged to the console.

To use authentication headers with the fetch function in JavaScript, you can pass an object with the headers property as the second argument to the fetch function. The headers property should be an object that contains the key-value pairs for the headers you want to include in the request.

For example, if you wanted to include an Authorization header with a bearer token,

you could do something like this:

```
const headers = {
  'Authorization': 'Bearer <your-bearer-token-here>'
};

fetch('https://example.com/api/v1/data', { headers })
  .then(response => response.json())
  .then(data => {
    // do something with the data here
  });
```

Listing 2.5: "Authroization header example with fetch

In this example, the headers object contains the Authorization header with a bearer token. This object is passed as the second argument to the fetch function, which includes the headers in the HTTP request.

Using the abort controller

The AbortController is a new API that allows you to abort an ongoing fetch() request. It is typically used when you want to cancel a request if the user navigates away from the current page, or if the user has started a new request that replaces the previous one.

Here is an example of how to use the AbortController with the fetch() method:

```
const controller = new AbortController();
const signal = controller.signal;

fetch('https://pokeapi.co/api/v2/pokemon/1', { signal })
  .then(response => response.json())
  .then(data => {
    console.log(data);
```

```
8       // do something with the data here
9     });
10
11  // later, if you want to cancel the request:
12  controller.abort();
```

Listing 2.6: AbortController example

Fetch vs AJAX

The main difference between fetch() and AJAX (Asynchronous JavaScript and XML) is that fetch() is a modern browser API, while AJAX is a technique used to send HTTP requests and retrieve data from a server. AJAX is based on the older XMLHttpRequest (XHR) API, which is supported by all modern browsers, but it has been largely replaced by the newer fetch() API.

Here are some other key differences between fetch() and AJAX:

- fetch() uses promises, while AJAX uses callbacks. This means that fetch() is easier to use and allows for more readable code, especially when dealing with asynchronous operations.
- fetch() supports the streaming of data, which means that you can start processing the data as soon as it becomes available, rather than having to wait for the entire response to be received. AJAX does not support streaming.
- fetch() supports the use of request and response objects, which provide a more powerful and flexible API for making web requests and handling responses. AJAX does not have this concept.

Overall, fetch() is a more modern and powerful API for making web requests, and it is the recommended way to perform HTTP requests in JavaScript.

2.3 Other ways to use fetch

The fetch function can be used for web scraping, but it is generally not the best option for this purpose. The fetch function is intended for making HTTP requests and retrieving data from a server, not for extracting data from an HTML page.

There are many dedicated tools and libraries that are better suited for web scraping, such as Puppeteer and Cheerio. These tools provide a more convenient and efficient way to extract data from HTML pages and can be easily integrated with the fetch function.

Here is an example of how you might use the fetch function and Cheerio to scrape data from an HTML page:

```
fetch('https://example.com')
  .then(response => response.text())
  .then(html => {
    const $ = cheerio.load(html);
    const data = $('#some-element').text();
    // do something with the data here
  });
```

Listing 2.7: Grabbing raw html with fetch and feeding that into a library

In this example, the fetch function is used to make a GET request to the example website, and then the response is passed to the then callback function. The response.text() method is used to convert the response to a string of HTML, which is then passed to Cheerio's load method. This creates a Cheerio object that can be used to extract data from the HTML using jQuery like syntax. In this case, the #some-element element is selected and its text content is extracted and stored in the data variable. From there, you can use the data however you like.

Again, this is just one example of how you might use the fetch function for web scraping. There are many other ways to accomplish this, and the specific approach you choose will depend on your specific needs and requirements.

2.3.1 Introduction to caching

A cache is a way of storing data so that future requests for the same data can be served faster. One way to use a cache with the fetch function (which is used to request data from a server) is to store the responses from fetch in a cache. Then, when a request is made for the same data, it can be served from the cache instead of making a new request to the server. This can improve the performance of your application by reducing the number of requests that need to be made to the server. When data is requested from a server, it can be stored in a cache so that future requests for the same data can be served faster. This is because the data can be served from the cache instead of making a new request to the server. This can improve the performance of the application by reducing the amount of time it takes to serve data to the user.

Caching can be especially beneficial in applications that make many requests to the same server, or in applications that are used by a large number of users who may be requesting the same data. In these cases, caching can reduce the load on the server and improve the overall performance of the application.

Here is an example of using a simple cache with the fetch function:

```
1 // Create a cache to store the responses from fetch
2 const cache = new Map();
3
4 // Define a function that uses fetch to request data from a server
5 function getData(url) {
```

```
  // Check if the data is already in the cache
  if (cache.has(url)) {
    // If it is, return the data from the cache
    return cache.get(url);
  } else {
    // If it's not in the cache, use fetch to request the data from
        the server
    return fetch(url)
      .then(response => response.json())
      .then(data => {
        // Store the data in the cache for future use
        cache.set(url, data);
        // Return the data
        return data;
      });
  }
}
```

In this example, the getData function uses fetch to request data from a server. If the data has been requested before, it will be served from the cache instead of making a new request to the server. This can improve the performance of your application by reducing the number of requests that need to be made to the server.

There are several JavaScript libraries that can be used to implement caching on the client side. Some examples include:

- lscache: This library is a simple in-memory cache that can be used to store data in the client's browser. It has a simple API and can be easily integrated into an application.
- Memoizee: This library is a simple utility that can be used to memoize (cache) the results of expensive function calls. It can be used to improve the perfor-

mance of an application by storing the results of frequently-used functions in a cache.

- QuickLRU: This library is a simple, lightweight, and efficient LRU (Least Recently Used) cache. It can be used to store data in a cache and automatically remove the least recently used items when the cache reaches its maximum size.

- tiny-lru: This library is a small and efficient LRU cache that can be used to store data in a cache. It has a simple API and is easy to integrate into an application.

These are just a few examples of JavaScript libraries that can be used to implement caching on the client side. There are many other libraries available, and the best one to use will depend on the specific requirements of your application.
There are more complete solutions like react-query that handle refreshing data, updating data as well as caching it.
To test your knowledge of async programming try to answer the following questions

1. Implement a post webhook to a discord channel

2. Parse the response from the pokeapi and return the url to the sprite of the master ball.

See page 67 for answers

3. Node

> JavaScript's global scope is like a public toilet. You can't avoid going in there, but try to limit your contact with surfaces when you do..
>
> — Dmitry Baranovskiy

To install Node.js on Windows, follow these steps:

- Go to the Node.js website: https://nodejs.org/
- Click the "Download" button to download the latest version of Node.js for Windows.
- Once the download is complete, run the installer and follow the on-screen instructions to install Node.js on your computer.
- Once the installation is complete, open a command prompt or terminal and type node -v to verify that Node.js was installed correctly and to see which version you have installed.

You can also use the following instructions to install Node.js using the Chocolatey package manager:

- Open a command prompt or terminal and run the following command:

```
choco install nodejs
```

- Once the installation is complete, type node -v to verify that Node.js was installed correctly and to see which version you have installed.

Alternatively, you can use the Windows Subsystem for Linux (WSL) to install and run Node.js on Windows. To do this, follow these steps:

- Enable the Windows Subsystem for Linux (WSL) feature on your computer. You can do this by opening the "Turn Windows features on or off" settings, scrolling down to the "Windows Subsystem for Linux" option, and checking the box next to it. Click "OK" to save the changes and enable WSL.
- Once WSL is enabled, open the Microsoft Store and search for "Linux". Select a Linux distribution, such as Ubuntu, and click "Get" to install it on your computer.
- Once the Linux distribution is installed, open a command prompt or terminal and type wsl to launch the Linux environment.
- In the Linux environment, follow the instructions for your specific distribution to install Node.js. For example, on Ubuntu, you can use the following command to install the latest version of Node.js:

```
sudo apt-get install nodejs
```

Once the installation is complete, you can use the node command to run Node.js in the Linux environment.

Alternatively, you can also use a package manager like apt on Ubuntu or brew on macOS to install Node.js. For example, on Ubuntu, you can use the following commands:

```
sudo apt update
sudo apt install nodejs
```

On macOS, you can use the following commands:

```
brew update
brew install node
```

These methods can provide additional benefits, such as automatic installation of dependencies and easier updates. Consult the documentation for your package manager for more information.

3.1 Package Managers

npm and Yarn are package managers for JavaScript. They are used to manage the dependencies (libraries and tools) that are required by a JavaScript project.

npm (short for Node Package Manager) is the default package manager for the JavaScript runtime environment Node.js. It is included with every Node.js installation, and is used to install and manage the packages (libraries and tools) that are required by a Node.js project. npm uses a registry (a database of available packages) to manage the packages that are available for download and installation.

Yarn is an alternative package manager for JavaScript that was developed by Facebook. It was created to address some of the limitations and challenges of using npm, such as slow installation times and difficulties managing multiple versions of

a package. Like npm, Yarn uses a registry to manage the packages that are available for download and installation. It also includes a variety of features that make it easier to manage dependencies, such as support for lockfiles and deterministic installs.

In summary, npm and Yarn are both tools that are used to manage the dependencies of a JavaScript project. They both use a registry to manage the available packages, but Yarn includes additional features that make it easier to manage dependencies and improve the performance of the installation process.

3.1.1 What is a package.json file

package.json is a file that is used in Node.js projects to define project metadata and specify the dependencies (libraries and tools) that are required by the project. It is typically located in the root directory of a Node.js project, and is used by the npm (Node Package Manager) to manage the project's dependencies.

The package.json file is a JSON (JavaScript Object Notation) file that contains a number of properties that define the metadata and configuration of the project. Some of the key properties of the package.json file include:

- name: The name of the project.
- version: The version of the project.
- scripts: A set of scripts that can be run using the npm run or yarn run command. For example, a start script might be defined to run the main entry point of the project.
- dependencies: A list of the dependencies (libraries and tools) that are required by the project. These dependencies will be installed when the npm install or

yarn install command is run.

- devDependencies: A list of the development dependencies (libraries and tools) that are required by the project, but are only needed in development (not in production). These dependencies will be installed when the npm install or yarn install command is run with the –dev flag.

In summary, the package.json file is a key file in a Node.js project. It defines the metadata and dependencies of the project, and is used by npm and Yarn to manage the project's dependencies.

3.2 Express

Express is a popular web application framework for building back-end applications with Node.js. It provides a simple and flexible way to create web servers and web applications, and includes a variety of features and tools that make it easier to develop and maintain back-end applications.
Some of the key features of Express include:

- A simple, lightweight, and flexible core that makes it easy to build web applications A routing system that allows you to define different routes for different HTTP methods and URLs

- Middleware support, which allows you to define functions that are executed before or after a request is handled by a route Built-in support for rendering HTML templates using popular template engines like Pug and EJS

- A large ecosystem of third-party libraries and plugins that can be easily integrated into Express applications.

- Express is widely used for building back-end applications because of its simplicity, flexibility, and rich feature set. It provides a solid foundation for building scalable and maintainable back-end applications with Node.js.

Here is an example of a simple Express server:

- In order to test save the file to app.js
- use yarn add express or npm install express, this should create a package.json file to track dependencies.
- run node app.js

```
1  const express = require("express");
2
3  const app = express();
4
5  app.get("/", (req, res) => {
6    res.send("Hello, world!");
7  });
8
9  app.listen(3000, () => {
10   console.log("Server listening on port 3000");
11 });
```

Listing 3.1: Simple Express Server

In this example, the Express app is created using the express function, and a route is defined for the / path that sends the string "Hello, world!" as a response. The app is then set to listen for incoming requests on port 3000. When a request is received on the / path, the specified response will be sent back to the client.

To add a start command to the package.json file that will run the app.js file, you can add a scripts property to the package.json file, and specify the start command as follows:

```
{
  "name": "my-node-app",
  "version": "1.0.0",
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    // Dependencies go here
    "express": "^5.0.0"
  }
}
```

In this example, the scripts property is added to the package.json file, and the start command is defined as node app.js. This means that when the start script is run (for example, by running npm start or yarn start), the app.js file will be executed using the node command.

Once the scripts property has been added to the package.json file, you can run the start script by using the npm run or yarn run command, followed by the name of the script. For example, to run the start script with npm, you can run the following command:

```
npm run start
```

To create an Express server that hosts static files, you can use the express.static middleware function. This function is part of the Express framework, which is a popular web application framework for Node.js.

Here is an example of how to use the express.static middleware to host static files:

```
const express = require('express');
const app = express();

app.use(express.static('public'));

app.listen(3000, () => {
  console.log('Server listening on port 3000');
});
```

Listing 3.2: Static files in express

In this example, the express.static middleware is used to serve all files in the public directory. This means that any file in the public directory can be accessed by a client by requesting the file's path relative to the public directory. For example, if there is a file named index.html in the public directory, a client can access it by making a request to http://your-server-domain/index.html.

You can also specify a different directory to serve static files from by passing the directory path as an argument to the express.static function. For example:

```
app.use(express.static('my-static-files'));
```

This will serve all files in the my-static-files directory.

It's important to note that the express.static middleware only serves static files. It does not handle dynamic requests, such as those made with a form or an AJAX request. To handle dynamic requests, you will need to use additional middleware or route handlers.

3.3 Docker

Docker is a popular platform for building, deploying, and running applications in containers. A container is a lightweight, standalone, and executable package of an application that includes everything the application needs to run, such as the code, libraries, and dependencies.

Containers provide a number of benefits over traditional approaches to building and deploying applications, such as virtual machines. Some of the key benefits of using Docker and containers include:

- Portability: Containers are isolated from the host system and can run on any machine that has Docker installed, regardless of the host operating system or system configuration. This makes it easy to deploy and run applications on different environments, such as local development, staging, and production.
- Consistency: Containers provide a consistent and predictable runtime environment for applications, which helps to ensure that applications will behave the same way no matter where they are deployed. This can reduce the time and effort required to troubleshoot issues and ensure that applications are working as expected.
- Scalability: Containers are lightweight and can be easily scaled up or down to meet the changing needs of an application. This makes it easy to manage the resources that an application uses, and allows applications to be quickly and easily adapted to handle changes in workload or traffic.
- Efficiency: Containers use less resources than traditional virtual machines, which makes it possible to run more applications on the same hardware. This

can help to reduce the cost and complexity of running and managing applications, and can improve the overall efficiency of an application's infrastructure.

In summary, Docker is a platform for building, deploying, and running applications in containers. Containers provide a number of benefits over traditional approaches, such as portability, consistency, scalability, and efficiency. This makes Docker a useful tool for building and deploying modern applications.

Here is an example of a Dockerfile that can be used to create a Docker image for a simple Express server:

```
1
2  Copy code
3  # Use the official Node.js 10 LTS image as the base image
4  FROM node:10-lts
5
6  # Create a directory for the application
7  RUN mkdir -p /usr/src/app
8
9  # Set the working directory to the application directory
10 WORKDIR /usr/src/app
11
12 # Copy the package.json and package-lock.json files to the working
       directory
13 COPY package*.json ./
14
15 # Install the dependencies
16 RUN npm install
17
18 # Copy the application source code to the working directory
19 COPY . .
20
```

```
21 # Expose port 3000 so it can be accessed from the outside
22 EXPOSE 3000
23
24 # Start the Express server when the container is started
25 CMD ["npm", "start"]
```

Listing 3.3: Dockerfile for node application

Your file directory should look like

```
/
├── package.json
├── package-lock.json
├── app.js
└── Dockerfile
```

In this Dockerfile, the official Node.js 10 LTS image is used as the base image. The package.json and package-lock.json files are copied to the working directory and the dependencies are installed using npm. The application source code is then copied to the working directory, and port 3000 is exposed so it can be accessed from the outside. Finally, the npm start command is specified as the default command for the container, which will start the Express server when the container is started.

To build a Docker image using this Dockerfile, you can run the following command from the directory where the Dockerfile is located:

```
docker build -t my-express-app .
```

This will build a Docker image with the name my-express-app using the instructions in the Dockerfile. You can then run the image as a Docker container using the following command:

```
docker run -p 3000:3000 my-express-app
```

This will start a Docker container based on the my-express-app image, and will

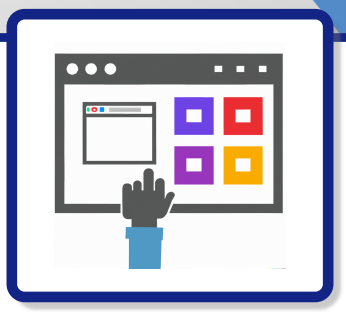
map port 3000 on the host to port 3000 in the container. The Express server will start automatically when the container is started, and you will be able to access the server at http://localhost:3000.

To test your knowledge please answer the following questions

Question 1 Make a docker file that uses node 16-lts, install packages with yarn and starts the server with yarn start.

See ?? for the answer.

4. Solutions to Excerises



Any application that can be written in JavaScript will eventually be written in JavaScript. — Jeff Atwood

---

Solution 1 Find the number that are greater than 10. See page 37

To filter an array for elements greater than 10 in JavaScript, you can use the Array.prototype.filter() method. This method takes a callback function as its argument, and returns a new array with only the elements from the original array that satisfy the condition specified by the callback function.

Here is an example of using Array.prototype.filter() to filter an array for elements greater than 10:

```
1 const numbers = [5, 10, 15, 20, 25];
2 // find value of greaterThan10
3 // const greaterThan10 = []
4 console.log(greaterThan10); // [15, 20, 25]
```

In this example, we have an array of numbers called numbers. We call the filter() method on numbers and pass a callback function that checks if the current element (num) is greater than 10. The filter() method will return a new array containing only the elements from numbers that are greater than 10 (in this case, 15, 20, and 25). The filter() method is a higher-order function, which means it takes a function as its input and returns a new function as its output. The callback function that you pass to filter() must take an element from the array as its input and return a Boolean.

Solution 2 See page 38 The output of the console.group is "' My Group This is the first log in my group This is the second log in my group undefined "' I am guessing you missed the undefined log at the end, it happens to the best of us, this is why its important to have a compiler on hand to see what the output will be.

Solution 3 See page 51 for details

To use the fetch function to send a message to a Discord webhook, you will need to do the following:

- Get the URL of the webhook from your Discord server settings. The URL will look something like this:

  `https://discordapp.com/api/webhooks/<webhook_id>/<webhook_token>.`

- Use the fetch function to send a POST request to the webhook URL. The fetch function takes the URL of the webhook as its first argument, and an object containing the request options (such as the HTTP method and the body of the request) as its second argument.

- In the request options, specify the HTTP method as POST and the body of

the request as a JSON object containing the message you want to send to the webhook. For example, the body of the request might look like this:

```
{
  "content": "This is a message sent via a Discord
  webhook using the fetch function."
}
```

- In the fetch function's promise, handle the response from the server to ensure that the message was sent successfully. For example, you might check the HTTP status code of the response to make sure it is in the 200 range, which indicates a successful response.

  Here is an example of using the fetch function to send a message to a Discord webhook:

```
1  // The URL of the webhook
2  const webhookUrl = "https://discordapp.com/api/webhooks/<
       webhook_id>/<webhook_token>";
3
4  // The body of the request
5  const requestBody = {
6    "content": "This is a message sent via a Discord webhook
         using the fetch function."
7  };
8
9  // Use fetch to send a POST request to the webhook
10 fetch(webhookUrl, {
11   method: "POST",
12   body: JSON.stringify(requestBody)
13 })
```

```
14  .then(response => {
15    // Check the HTTP status code of the response
16    if (response.status >= 200 && response.status < 300) {
17      console.log("Message sent successfully!");
18    } else {
19      console.error("Error sending message:", response.
            statusText);
20    }
21  })
22  .catch(error => {
23    console.error("Error sending message:", error);
24  });
```

Listing 4.1: Post to webhook url using fetch

In order to implement this in node you may have to install node-fetch.

Solution 4 See page 51 for the question

To use the fetch function to make a request to the PokeAPI and parse the results, you can do the following:

- Define a function that takes the URL of the API endpoint as an argument. In this case, the URL would be https://pokeapi.co/api/v2/item/1.
- Use the fetch function to make a GET request to the API endpoint. The fetch function takes the URL of the endpoint as its first argument, and an object containing the request options (such as the HTTP method and any request headers) as its second argument.
- In the fetch function's promise, use the json method of the response object to parse the JSON data returned by the API. This will return a JavaScript object

containing the data from the API.

- Access the properties of the parsed data object to get the information you need. For example, if you want to get the name of the item, you could access the name property of the data object.

Here is an example of using the fetch function to make a request to the PokeAPI and parse the results:

```
// Define a function that makes a request to the PokeAPI
function getItem(url) {
  // Use fetch to make a GET request to the API endpoint
  return fetch(url, {
    method: "GET"
  })
    .then(response => {
      // Parse the JSON data returned by the API
      return response.json();
    })
    .then(data => {
      // Access the properties of the data object
      console.log("Item name:", data.name);
      console.log("Item description:", data.description);
      // Return the data object
      return data;
    });
}

// Call the function to make the request
getItem("https://pokeapi.co/api/v2/item/1").then(data => {
  // Do something with the data
  console.log("Item data:", data);
```

```
24 });
```

Listing 4.2: Fetch data from Pokeapi

Afterwards we can parse the result to grab the url to the sprite for the master ball.

```
getItem("https://pokeapi.co/api/v2/item/1").then(data => {
  // Do something with the data
  console.group("POKEAPI data")
  console.log("Item data:", data);
  const masterBall= data.sprites.default;
  console.log("masterBall", masterBall);
  console.groupEnd();
});
```

Listing 4.3: "parsing result from pokeapi

Solution 5 Here is an example of a Dockerfile that uses Node 16-LTS, installs packages with Yarn, and starts the server with yarn start:

```
# Use the Node 16-LTS image as the base image
FROM node:16-lts

# Set the working directory to the app directory
WORKDIR /app

# Copy the package.json and yarn.lock files to the app directory
COPY package.json yarn.lock ./

# Install the app dependencies with Yarn
RUN yarn install

# Copy the rest of the app's source code to the app directory
```

```
14 COPY . .
15
16 # Expose the app's port
17 EXPOSE 3000
18
19 # Start the app with Yarn
20 CMD ["yarn", "start"]
```

This Dockerfile does the following:

- Uses the node:16-lts image as the base image
- Sets the working directory to the /app directory
- Copies the package.json and yarn.lock files to the /app directory
- Installs the app dependencies with Yarn
- Copies the rest of the app's source code to the /app directory
- Exposes the app's port (3000 in this example)
- Starts the app with yarn start

Once you have created this Dockerfile, you can build a Docker image by running the docker build command and specifying the path to the Dockerfile. For example:

```
docker build -t my-app .
```

This will create a Docker image named my-app based on the instructions in the Dockerfile. You can then run this image as a container using the docker run command: See page 64 for the question

Code Snippets

Bibliography

[1] Mdn. *JavaScript — Dynamic client-side scripting*. 2022. URL: https://developer.mozilla.org/en-US/docs/Learn/JavaScript (visited on 11/30/2022).

[2] mdn. *Glossary MDN Falsy*. 2022. URL: https://developer.mozilla.org/en-US/docs/Glossary/Falsy (visited on 11/30/2022).

[3] Project Pro. *10 reasons why you should use NodeJs*. 2022. URL: https://www.projectpro.io/article/10-reasons-why-you-should-use-nodejs/129 (visited on 11/30/2022).

Index