Javascript Cheatsheet - Basics

Data Types

JavaScript has six primitive data types:

1. Number: This represents numeric values, such as `3`, `3.14`, or `NaN` (Not a Number).
2. String: This represents textual data, enclosed in single or double quotes, such as `"hello"`, `'world'`, or `"123"`.
3. Boolean: This represents logical values, `true` or `false`.
4. Undefined: This represents a value that is not yet defined or has no value.
5. Null: This represents an intentional absence of any object value.
6. Symbol (new in ECMAScript 6): This data type represents a unique identifier that is not equal to any other identifier.

JavaScript also has one non-primitive data type:

7. Object: This represents a collection of key/value pairs, which can be used to store various data types and complex entities.

It's worth noting that JavaScript is a dynamically-typed language, which means that the data type of a variable is determined at runtime based on the type of value that it currently holds. This is in contrast to statically-typed languages where the data type of a variable must be explicitly declared and is fixed at compile-time.

Write a Function/Define a function

To write a function in JavaScript, you can use the `function` keyword followed by the name of the function, and then include the code that you want the function to execute inside curly braces `{}`.

Example

Here is a basic example:

```
function greet(name) {
  console.log(`Hello, ${name}!`);
}
```

In this example, we have defined a function called `greet` that takes a single argument `name`. When the function is called with an argument, it will log a message to the console that greets the person by name.

> NOTE: Sometimes there is no argument for the function. Here is an example of what it would look like: `function greet() {};`

Call a function/Use a function

To call a function in JavaScript, you can simply write the name of the function followed by parentheses `()`, and include any arguments that the function requires inside the parentheses. Here is an example of calling the `greet` function with an argument:

Example

```
greet("Alice"); // logs "Hello, Alice!"
```

This will call the `greet` function with the argument `"Alice"`, which will log the message `"Hello, Alice!"` to the console.

> NOTE: Here is how you would call a function with no arguments: `greet();`

Arrays

An array is a collection of elements (or values) of the same data type, arranged in a contiguous block of memory and identified by a single name. You can think of an array as a group of boxes or slots that hold values of the same type, where each box is identified by an index or a position number.

Arrays are used to store and manipulate large amounts of data, such as lists of numbers, names, or other objects.

Example

```
let items = [];
```

The above sets a empty array to a variable named items.

.push()

`.push()` is a method that is used to add one or more elements to the end of an array. It modifies the original array and returns the new length of the array after the new elements have been added.

Example

```
items.push('brush')
```

`items` is the name of the array.

`'brush'` is the value we want to add to the `items` array. In this case, we want to add a string to the array.

> **NOTE:** Typically, we will create a variable that we want to push to the array.

An example that defines a variable we want to add to an array.

Define a variable and set it equal to value. We are defining a string with a value of `'brush'` in this example.

```
let itemToAdd = 'brush';
```

Then, we can add this item to the items array:

```
items.push(itemToAdd)
```

This will add 'brush' to the end of the `items` array.

.forEach()

`forEach()` is an array method that is used to iterate over the elements of an array and execute a provided callback function once for each element.

Example

```
let numbers = [1, 2, 3];

numbers.forEach(function(number) {
  console.log(number);
});

// Output:
// 1
// 2
// 3
```

`let numbers = [1, 2, 3];`

`numbers` is the array. In this example, it's an array of number 1, 2, 3

```
numbers.forEach(function(number) {
  console.log(number);
});
```

`numbers` is the array we want to loop through

`function(number){}` is the callback. This represents what we want to happen next to each value of the array. The `number` represents one item in the array. Typically, this value is a singular version of what the array is called. In this example, since we have an array called `numbers` plural, then we use `number` to identify each value of the array.

Think of this as we are going through each value in the array, `number` is one value in the array. The `console.log(number)` is displaying each value in the array, one at a time. Essentially, we want to grab one item in the array, do something to that item (in this example, we are using `console.log(number)` to display the number) and the do the same thing to the next item in the array until we reach the end of the array.

.find()

Example from practice demo project:

`const todo = todos.find(todo => todo.id === parseInt(event.target.id));`

Here is a breakdown of what each part of the code does:

- `const todo = ...` creates a new constant variable todo and assigns it the value of the first element in the todos array that satisfies the condition defined by the callback function.
- `todos` is an array of objects that represent tasks in a to-do list.
- `.find()` is an array method that searches the array for the first element that satisfies a certain condition, and returns that element. If no element satisfies the condition, undefined is returned.
- `todo => todo.id === parseInt(event.target.id)` is a callback function that is passed to the `find()` method as an argument. This function takes one argument, `todo`, which represents each element of the `todos` array that is being iterated over. The function compares the `id` property of each `todo` object with the `id` property of the HTML element that triggered the event (e.g. a button click).
- `parseInt(event.target.id)` converts the `id` property of the HTML element that triggered the event (which is a string) to an integer using the `parseInt()` function. This is necessary because the `id` property of the `todo` object is also an integer.
- `todo.id === parseInt(event.target.id)` compares the `id` property of the `todo` object with the parsed `id` property of the HTML element that triggered the event. If they are equal, the comparison will return `true`, and the `find()` method will return the first element in the todos array that satisfies the condition.

.findIndex()

Example from practice demo project: `const todoIndex = todos.findIndex(todo => todo.id === parseInt(event.target.id));`

This line of code is using the `findIndex()` method to search for the index of a specific todo item in an array called `todos`.

The `findIndex()` method returns the index of the first element in the array that satisfies a provided testing function. In this case, the testing function is `todo => todo.id === parseInt(event.target.id)`, which takes a single parameter `todo` and returns a Boolean value indicating whether the `id` property of `todo` matches the `id` property of the element that triggered the event. The `parseInt()` function is used to convert the `id` property (which is typically a string) to an integer so that it can be compared to the `id` of the event target.

Once the `findIndex()` method finds the first element that satisfies the testing function, it returns the index of that element in the array. If no element in the array satisfies the testing function, `findIndex()` returns -1.

The returned index is then assigned to the constant variable `todoIndex`, which can be used to update or remove the corresponding todo item from the `todos` array.

Variable

A variable is a named storage location that holds a value. You can think of a variable as a container that holds data, like a box that holds objects.

When you define a variable, you give it a name and a type, and you can optionally assign an initial value to it.

Example

```
let message = 'Hello World!';
```

You define a variable using the `let` keyword

message is the name of the variable.

The equals sign is necessary to set a value to the variable.

'Hello World!' is the value of the variable. In this example, it is a string but it can be any type.

Constant Variable

A variable that is assigned a value once and cannot be changed throughout the program's execution. In other words, once a constant variable is initialized, its value cannot be modified.

Constant variables are useful for situations where you want to declare a value that should not be changed by mistake. By declaring a variable as a constant, you can prevent accidental modification of its value and make your code more robust and error-resistant.

Examples with different data types

Example One:

`const defaultNumber = 700;`

You define a constant variable using the `const` keyword.

defaultNumber is the name of the variable.

The equals sign is necessary to set a value to the variable.

700 is the value of the variable. In this example, it is a integer but it can be any type.

Example Two:

```
const defaultObject = {
    id: 848454,
    type: '',
    completed: false,
};
```

You define a constant variable using the `const` keyword.

defaultObject is the name of the constant variable.

The equals sign is necessary to set a value to the variable.

Since this constant variable is an object, we use the curly braces `{}` to denote an object type.

> **NOTE** An object is a collection of key-value pairs, where each key represents a property name and each value represents the value of that property. You can think of an object as a dictionary or a map, where each property name is a key that maps to a value.

The key-values consist of the following:

- `id` is the key, the `id` is set to a value of `848454`, `848454` is an integer
- `type` is the key, the `type` is set to a value of `''`, `''` is an empty string
- `completed` is the key, the `completed` is set to a value of `false`, `false` is an boolean

> **NOTE** the boolean data type represents a logical value that can be either true or false. It is used to represent the truth value of an expression or a condition, where true represents a condition that is true or valid, and false represents a condition that is false or invalid.

Built in Javascript Functions

createElement()

`document.createElement()` is a JavaScript method that creates a new HTML element, which can be later inserted into the DOM tree.

Example

Here is an example of how to use `document.createElement()` to create a new `<div>` element:

`const newDiv = document.createElement("div");`

In this example, `document.createElement("div")` creates a new `<div>` element and assigns it to the `newDiv` constant variable. At this point, the new `<div>` element exists only in memory and is not yet part of the DOM tree.

Once you have created a new element using `document.createElement()`, you can manipulate its attributes, content, and style, and append it to other elements in the DOM tree using methods like `appendChild()`, `setAttribute()`, and `innerHTML()`.

For example, to set the `id` attribute of the new `<div>` element to "my-div" and add some text content to it, you can use the following code:

```
newDiv.setAttribute("id", "my-div");

newDiv.innerHTML = "Hello, world!";
```

Once you have finished manipulating the new element, you can add it to the DOM tree by appending it to an existing element using `appendChild()`, like this:

`document.body.appendChild(newDiv);`

This will add the new `<div>` element as a child of the `<body>` element, and it will now be visible in the rendered HTML page.

appendChild()

`appendChild()` is a JavaScript function that is used to add a new child element to an existing parent element in the HTML Document Object Model (DOM). It takes a single argument, which is the child element that you want to add.

The syntax for using `appendChild()` is as follows:

```
parentElement.appendChild(childElement);
```

Here, `parentElement` is the existing parent element to which you want to add the new child element, and `childElement` is the new child element that you want to add.

Example

Here's an example of using `appendChild()` to add a new `li` element to an existing `ul` element:

```
let ul = document.getElementById("myList");

let li = document.createElement("li");

li.textContent = "New item";

ul.appendChild(li);
```

In this example, we first used `document.getElementById()` to select an existing `ul` element with the ID "myList". We then used `document.createElement()` to create a new `li` element, and set its `textContent` property to "New item". Finally, we used `appendChild()` to add the new `li` element as a child of the existing ul element.

splice()

`splice()` is a JavaScript method that allows you to add or remove elements from an array. It modifies the array in place and returns an array of the removed elements (if any).

The `splice()` method takes two or more arguments:

- The index at which to start changing the array
- The number of elements to remove (if any)
- Zero or more elements to insert into the array at the specified index

Example

```
let fruits = ["apple", "banana", "orange", "kiwi"];

fruits.splice(2, 1);

// The above line will remove one element starting at index 2 (orange)

console.log(fruits); // ["apple", "banana", "kiwi"]
```

In this example, we first created an array of fruits. We then used `splice()` to remove one element from the array starting at index 2, which is the element "orange". The resulting array contains the remaining elements: ["apple", "banana", "kiwi"].

Javascript Properties

innerHTML

`innerHTML` is a JavaScript property that is used to **get** or **set** the HTML content of an element in the Document Object Model (DOM).

When used to get the content of an element, `innerHTML` returns a string that represents the content of the element, including any HTML tags that are present.

When used to set the content of an element, `innerHTML` replaces any existing content in the element with the new content that is specified. This can include plain text, HTML tags, or a combination of both.

Example

Here's an example of using `innerHTML` to **set** the content of an element:

```
let div = document.getElementById("myDiv");

div.innerHTML = "<p>This is some new content</p>";
```

In this example, we first used document.getElementById() to select an existing div element with the ID "myDiv". We then used the innerHTML property to set the content of the element to a new paragraph element containing the text "This is some new content".

Conditionals

If, Else, Else If Conditionals

`if` is a conditional statement that allows you to execute code based on a specified condition. The syntax for an if statement is as follows:

```
if (condition) {
  // code to execute if condition is true
}
```

In this syntax, condition is any expression that can be evaluated as either true or false. If condition is true, the code inside the curly braces will be executed. If condition is false, the code inside the curly braces will be skipped.

Example

```
let x = 5;

if (x > 0) {
  console.log("x is positive");
}
```

In this example, we first assigned the value 5 to the variable `x`. We then used an `if` statement to check whether `x` is greater than 0. Since 5 is indeed greater than 0, the code inside the `if` block will be executed, which logs the message "x is positive" to the console.

`if` statements are a fundamental feature of JavaScript and are used extensively in programming to make decisions based on specific conditions. They can be combined with other control structures, such as `else` and `else if`, to create more complex decision-making logic in your code.

Else If conditional

an `else if` statement is used to add additional conditions to an `if` statement. It allows you to test multiple conditions and execute different code based on the result of those tests.

```
if (condition1) {
  // code to execute if condition1 is true
} else if (condition2) {
  // code to execute if condition2 is true
} else if (condition3) {
  // code to execute if condition3 is true
} else {
  // code to execute if none of the conditions are true
}
```

In this syntax, you start with an `if` statement and specify a condition to test. If the condition is true, the code inside the corresponding block will be executed. If the condition is false, the program moves on to the next else if statement.

If none of the conditions in the `if` or `else if` statements are true, the code inside the else block will be executed.

Example

```
let x = 5;

if (x > 0) {
  console.log("x is positive");
} else if (x < 0) {
  console.log("x is negative");
} else {
  console.log("x is zero");
}
```

In this example, the program first checks whether `x` is greater than 0. Since 5 is indeed greater than 0, the code inside the first block will be executed, and the message "x is positive" will be logged to the console. If `x` had been negative, the code inside the second block would have been executed, and if `x` had been 0, the code inside the `else` block would have been executed.

Event Listener

`addEventListener()` - a JavaScript method that allows you to register an event listener on an HTML element. The event listener will respond to events that occur on the element, such as a click or a keypress, by executing a specified function.

The addEventListener() method takes two arguments:

The type of event to listen for (e.g. "click", "keypress", etc.) The function to execute when the event occurs.

Example

```
let button = document.getElementById("myButton");

button.addEventListener("click", function() {
  console.log("Button clicked!");
});
```

In this example, we first used `document.getElementById()` to select an existing button element with the ID "myButton". We then used `addEventListener()` to register a click event listener on the button. The listener is an anonymous function that logs the message "Button clicked!" to the console when the button is clicked.

Event Target Checked

`event.target.checked` is a Boolean property that indicates whether a checkbox or radio button is checked or not, and is accessed through the `target` property of an event object.

In HTML, checkboxes and radio buttons are input elements with a `type` attribute of "checkbox" or "radio", respectively. When the user interacts with a checkbox or radio button by clicking on it, a `"change"` event is fired, and the event object contains information about the element that triggered the event.

By accessing the `event.target.checked` property, you can determine whether the checkbox or radio button that triggered the event is currently checked or not. The `event.target` property refers to the HTML element that triggered the event, and `checked` is a property of input elements of type "checkbox" and "radio".

Example

Here is an example of how to use `event.target.checked` to perform an action based on the checked state of a checkbox:

```
const myCheckbox = document.querySelector("#my-checkbox");

myCheckbox.addEventListener("change", function(event) {
  if (event.target.checked) {
    console.log("Checkbox is checked!");
  } else {
    console.log("Checkbox is not checked!");
  }
});
```

In this example, the `addEventListener()` method is used to listen for a `"change"` event on a checkbox element with the `id` "my-checkbox". When the event is fired, the callback function checks the `checked` property of `event.target` to determine whether the checkbox is currently checked or not, and logs a message to the console accordingly.