JavaScript Cheatsheet

A comprehensive cheat sheet covering essential JavaScript concepts, syntax, and best practices for developers of all levels.

JavaScript Fundamentals

Variables and Data Types

`var`	Oldest way to declare a variable. Function-scoped or globally-scoped.
`let`	Block-scoped variable declaration. Preferred for variables that may be re-assigned.
`const`	Block-scoped constant declaration. Value cannot be re-assigned after initialization.
Primitive Data Types	String, Number, Boolean, Null, Undefined, Symbol, BigInt
Objects	Collections of key-value pairs. Keys are strings, values can be any data type.
Arrays	Ordered lists of values.

Operators

Arithmetic	`+`, `-`, `*`, `/`, `%`, `**` (exponentiation)
Assignment	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `**=`
Comparison	`==` (equal), `!=` (not equal), `===` (strict equal), `!==` (strict not equal), `>`, `<`, `>=`, `<=`
Logical	`&&` (and), `||` (or), `!` (not)
Ternary	`condition ? expr1 : expr2`
typeof	returns a string indicating the type of the unevaluated operand.

Control Flow

`if...else` statement:

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

`switch` statement:

```
switch (expression) {
  case value1:
    // code to execute if expression === value1
    break;
  case value2:
    // code to execute if expression === value2
    break;
  default:
    // code to execute if expression doesn't match any case
}
```

`for` loop:

```
for (initialization; condition; increment) {
  // code to execute repeatedly
}
```

`while` loop:

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

`do...while` loop:

```
do {
  // code to execute at least once
} while (condition);
```

`for...in` loop:

```
for (key in object) {
  // code to execute for each property in object
}
```

`for...of` loop:

```
for (element of iterable) {
  // code to execute for each element in iterable
}
```

Functions and Objects

Functions

Function Declaration	`function functionName(parameters) { // code to execute return value; }`
Function Expression	`const functionName = function(parameters) { // code to execute return value; };`

Arrow Function	`const functionName = (parameters) => { // code to execute return value; };`
	
	Implicit return:
	`const functionName = param => param * 2;````

`this` keyword	Refers to the object it belongs to. Its value depends on how the function is called.
Callback Functions	Functions passed as arguments to other functions.
Closures	Functions that have access to variables from their outer scope, even after the outer function has finished executing.

Objects

Object Literal	`const object = { key1: 'value1', key2: 'value2', method: function() { } };`
Accessing Properties	`object.key1` or `object['key1']`
Adding/Modifying Properties	`object.newKey = 'newValue';`
Deleting Properties	`delete object.key1;`

Object Constructor	`function MyObject(prop1, prop2) { this.prop1 = prop1; this.prop2 = prop2; }`
	
	`const obj = new MyObject('value1', 'value2');`

Classes	`class MyClass {`
	
	`constructor(prop1, prop2) { this.prop1 = prop1; this.prop2 = prop2; }`
	
	`method() { } }`
	
	`const obj = new MyClass('value1', 'value2');`

Arrays

Creating Arrays:

```
const arr = [1, 2, 3];
const arr2 = new Array(1, 2, 3); // Avoid this
```

Array Methods:

- `push(element)`: Adds an element to the end.
- `pop()`: Removes the last element.
- `shift()`: Removes the first element.
- `unshift(element)`: Adds an element to the beginning.
- `slice(start, end)`: Returns a portion of the array.
- `splice(start, deleteCount, ...items)`: Adds/removes elements.
- `concat(arr2)`: Combines arrays.
- `join(separator)`: Converts array to a string.
- `indexOf(element)`: Returns the first index of an element.
- `includes(element)`: Checks if an array contains an element.
- `find(callback)`: Returns the first element that satisfies a condition.
- `findIndex(callback)`: Returns the index of the first element that satisfies a condition.
- `filter(callback)`: Creates a new array with elements that satisfy a condition.
- `map(callback)`: Creates a new array by applying a function to each element.
- `reduce(callback, initialValue)`: Reduces the array to a single value.

DOM Manipulation

Selecting Elements

`document.getElementById(id)`	Selects an element by its ID.
`document.querySelector(selector)`	Selects the first element that matches a CSS selector.
`document.querySelectorAll(selector)`	Selects all elements that match a CSS selector (returns a NodeList).
`document.getElementsByClassName(className)`	Selects all elements with a given class name (returns an HTMLCollection).
`document.getElementsByTagName(tagName)`	Selects all elements with a given tag name (returns an HTMLCollection).

Modifying Elements

`element.innerHTML`	Gets or sets the HTML content of an element.
`element.textContent`	Gets or sets the text content of an element.
`element.setAttribute(name, value)`	Sets the value of an attribute on an element.
`element.getAttribute(name)`	Gets the value of an attribute on an element.
`element.classList.add(className)`	Adds a class to an element.
`element.classList.remove(className)`	Removes a class from an element.
`element.classList.toggle(className)`	Toggles a class on an element.

Events

Adding Event Listeners:

```
element.addEventListener(event, function, useCapture);
```

Common Events:

- `click`: Mouse click.
- `mouseover`: Mouse pointer is over the element.
- `mouseout`: Mouse pointer leaves the element.
- `keydown`: Key is pressed down.
- `keyup`: Key is released.
- `submit`: Form submission.
- `load`: Page has finished loading.
- `DOMContentLoaded`: DOM has finished loading.

Example:

```
const button = document.querySelector('button');
button.addEventListener('click', function(event) {
  console.log('Button clicked!', event);
});
```

Asynchronous JavaScript

Callbacks

Functions that are executed after another function has finished executing.

Example:

```
function fetchData(callback) {
  setTimeout(() => {
    const data = 'Data fetched!';
    callback(data);
  }, 1000);
}

fetchData(function(data) {
  console.log(data); // Output: Data fetched!
});
```

Promises

Creating a Promise	`const promise = new Promise((resolve, reject) => { // Asynchronous operation if (/* operation successful */) { resolve(value); } else { reject(error); } });`
Handling a Promise	`promise .then(value => { // Handle successful result }) .catch(error => { // Handle error }) .finally(() => { // Code that always executes });`
Promise.all()	`Promise.all([promise1, promise2]) .then(results => { // results is an array of the resolved values });`
Promise.race()	`Promise.race([promise1, promise2]) .then(result => { // result is the resolved value of the first promise to resolve });`

Async/Await

Syntactic sugar for working with promises, making asynchronous code look and behave a bit more like synchronous code.

```
async function myFunction() {
  try {
    const result = await someAsyncFunction();
    console.log(result);
  } catch (error) {
    console.error(error);
  }
}
```