Summary of JavaScript Basics

Notes by Taslim Ansari

1. Introduction to JavaScript

- **JavaScript** is a **high-level interpreted programming language** commonly used for web development.
- Enables dynamic and interactive website features such as animations, form validation, and real-time updates.
- Also utilized for server-side programming and mobile app development.
- Difference between **interpreted** (JavaScript) and **compiled** languages (like C++, Rust): interpreted languages execute code line-by-line, whereas compiled languages are converted into machine code beforehand.
- High-level languages offer **human-readable syntax** and abstract away hardware complexities.

2. Setting Up the Environment

- Uses **Google Chrome Developer Tools Console** to write and test JavaScript code.
- Instructions on opening and navigating the console are provided.

Variables and Data Types

Variable Declaration

- Three main keywords: `var`, `let`, and `const`.
  - `var`: Function-scoped, older style.
  - `let`: Block-scoped, variable can be reassigned.
  - `const`: Block-scoped, value cannot be reassigned.
- Example:

```
var name = "John Doe";
let age = 30;
```

```
const country = "USA";
```

Data Types

- **Primitive Types:**
  - `number`: integers, floating points, negatives.
  - `string`: sequences of characters, enclosed in single or double quotes.
  - `boolean`: `true` or `false`.
  - `null`: explicitly set empty value.
  - `undefined`: variable declared but not assigned.
- Important distinctions:
  - `null !== undefined` (not strictly equal).
  - Type coercion can lead to implicit conversions and unexpected results.

Type Coercion and Comparison

- JavaScript tries to be "nice" by converting types implicitly during operations (e.g., adding `1 + "2"` results in `"12"` string).
- Two equality operators:
  - `==` (loose equality, allows coercion).
  - `===` (strict equality, checks type and value).
- Use strict equality (`===`) to avoid bugs from type coercion.

Numeric Precision

- Floating point arithmetic can produce imprecise results (e.g., `0.1 + 0.2 !== 0.3`).
- The `.toFixed()` method can help manage precision by formatting numbers.

Functions

- Defined using the `function` keyword or as **Arrow functions** (shorthand syntax).
- Functions can accept parameters and return values.
- Functions are **first-class citizens** in JavaScript:

    - Can be assigned to variables.
    - Passed as arguments.
    - Returned from other functions.
- Example of function assignment:

```
const sum = function(a, b) { return a + b; };
let x = sum;
x(5, 4); // returns 9
```

- Arrow function example:

```
const square = x => x * x;
```

Arrays and Array Methods

Arrays

- Ordered collections indexed from zero.
- Elements accessed or modified using square bracket syntax.
- Example:

```
let letters = ['a', 'b', 'c', 'd', 'e'];
```

Manipulating Arrays

Method	Purpose	Returns
`pop()`	Removes last element	The removed element
`push(element)`	Adds element to end	New length of the array
`shift()`	Removes first element	The removed element
`unshift(element)`	Adds element to beginning	New length of the array

- Arrays can contain mixed data types, including numbers and strings.

Spread Operator

- Syntax: `...array`

- Used to create new arrays by expanding existing arrays, avoiding nested arrays.
- Example:

```
let newArr = ['a', ...letters, 5];
```

Array Iteration Methods

- `filter(predicate)` returns a new array of elements that satisfy the predicate function.
- `map(callback)` creates a new array with each element transformed by the callback.
- `reduce(callback, initialValue)` reduces the array to a single value by cumulatively applying the callback.

Example for filtering even numbers:

```
const evenNumbers = numbers.filter(x => x % 2 === 0);
```

Example for mapping to double values:

```
const doubled = numbers.map(x => x * 2);
```

Example for reducing to sum:

```
const total = ages.reduce((acc, x) => acc + x, 0);
```

Objects and Object Methods

Object Basics

- Collections of key-value pairs.
- Keys (property names) are strings; values can be any data type.
- Access properties via dot notation or brackets:

```
person.name;
```

```
person['age'];
```

- Properties can be added or deleted dynamically.
- Example object:

```
let person = { name: "John Doe", age: 30, country: "USA" };
```

Methods

- Object properties that are functions are called **methods**.
- Methods can use the special `this` keyword to refer to the object itself.
- Example:

```
person.welcome = function() {
  console.log("Hello " + this.name);
};
person.welcome(); // "Hello John Doe"
```

Useful Object Methods

Method	Description	Returns
`Object.keys(obj)`	Returns an array of all property names	Array of strings
`Object.values(obj)`	Returns an array of all property values	Array of values

- These methods enable iterating over object properties using loops.

Control Structures

Conditional Statements

- `if`, `else if`, `else` control execution based on conditions.
- Example:

```
if(score >= 80) {
  console.log("Great");
} else if(score >= 60) {
  console.log("Good enough");
```

```
} else {
  console.log("Failed");
}
```

Loops: For Loop

- Used to iterate a fixed number of times, commonly over arrays.
- Syntax includes initialization, condition, and increment.
- Example iterating array elements:

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

Variable Scope

Keyword	Scope Type	Scope Description
`var`	Function scope	Variable accessible anywhere inside the function it is declared in
`let`	Block scope	Variable accessible only within the nearest enclosing block
`const`	Block scope	Same as `let` but value cannot be reassigned

- Block: any code enclosed in `{}` such as if statements, loops.
- Function scope means variables are accessible throughout the entire function.
- Block scope means variables exist only inside blocks.

Key Insights

- **JavaScript is versatile**, used for front-end, back-end, and mobile development.
- **Understanding variable declarations and scope is fundamental** to avoid bugs.
- **Type coercion can cause subtle bugs**; prefer strict equality (`===`).

- **Functions are first-class citizens**, enabling functional programming patterns.
- **Arrays and objects in JavaScript are objects with methods and properties**, allowing rich data manipulation.
- **Higher-order functions like `filter`, `map`, and `reduce` simplify array processing**.
- **Arrow functions provide concise syntax**, beneficial for callbacks and functional methods.
- **Use of `this` in methods allows objects to encapsulate behavior and data**.
- **Control structures and loops control program flow effectively**.
- **Spread operator facilitates immutable array manipulation**.

Quantitative Data: Variable Scope and Behavior

Variable Type	Scope	Can Reassign?	Example Behavior Summary
`var`	Function	Yes	Accessible throughout function, same variable in nested blocks
`let`	Block	Yes	Accessible only within block, different variables in different blocks
`const`	Block	No	Same as `let` but value cannot be changed

Hoisting

- **Hoisting** is JavaScript's behavior of moving **declarations** (not initializations) to the top of their scope during execution.
- Affects variables declared with `var`, `let`, `const`, and function declarations.

Hoisting with `var`

- Variables declared with `var` are hoisted and initialized with `undefined`.

Example:

```
console.log(a); // undefined
```

```
var a = 10;
```

Hoisting with `let` and `const`

- `let` and `const` are hoisted but **not initialized**.
- They exist in the **Temporal Dead Zone (TDZ)** until the declaration line is executed.
- Accessing them before declaration throws an error.

Example:

```
console.log(b); // ReferenceError
let b = 20;
```

Hoisting with Functions

- Function declarations are fully hoisted and can be called before their definition.

Example:

```
sayHello();

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

Truthy and Falsy Values

- JavaScript evaluates values as **truthy** or **falsy** when used in conditions.
- **Falsy values** are treated as `false` in conditional statements.

Falsy Values in JavaScript

- `false`
- `0`
- `""` (empty string)

- `null`
- `undefined`
- `NaN`
- All other values are considered **truthy**.

Example:

```
if (" ") {
  console.log("This runs"); // truthy
}
```

Array Iteration: forEach vs map vs filter

Method	Returns New Array	Use Case
`forEach()`	No	Perform side effects like logging or updating values
`map()`	Yes	Transform each element in an array
`filter()`	Yes	Select elements based on a condition

Example:

```
numbers.forEach(x ⇒ console.log(x));
const squared = numbers.map(x ⇒ x * x);
const positives = numbers.filter(x ⇒ x > 0);
```

for...of and for...in Loops

for...of Loop

- Used to iterate over **iterable objects** such as arrays.
- Accesses values directly.

Example:

```
for (let letter of letters) {
  console.log(letter);
```

```
}
```

for...in Loop

- Used to iterate over **object properties (keys)**.
- Accesses property names.

Example:

```
for (let key in person) {
  console.log(key, person[key]);
}
```

Asynchronous JavaScript

- JavaScript is **single-threaded**, meaning it executes one task at a time.
- Asynchronous programming allows long-running tasks (like API calls) to run without blocking the main thread.

Promises

- A **Promise** represents a value that may be available now, later, or never.
- Has three states: `pending`, `fulfilled`, `rejected`.

Example:

```
fetch(url)
  .then(response ⇒ response.json())
  .then(data ⇒ console.log(data))
  .catch(error ⇒ console.log(error));
```

---

async / await

- `async` makes a function return a Promise.
- `await` pauses execution until the Promise resolves.
- Provides cleaner, more readable asynchronous code.

Example:

```
async function getData() {
  try {
    const response = await fetch(url);
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.log(error);
  }
}
```

Error Handling

- JavaScript uses `try...catch` blocks to handle runtime errors gracefully.
- Prevents the program from crashing due to unexpected errors.

Example:

```
try {
  let data = JSON.parse("{invalid}");
} catch (error) {
  console.log("Error occurred");
}
```

JSON (JavaScript Object Notation)

- **JSON** is a lightweight data format used for data exchange between client and server.
- Commonly used in APIs.

JSON Methods

Method	Purpose
`JSON.stringify()`	Converts JavaScript object to JSON string
`JSON.parse()`	Converts JSON string to JavaScript object

Example:

```
const jsonString = JSON.stringify(person);
const obj = JSON.parse(jsonString);
```

Primitive vs Reference Types

Primitive Types

- Stored by **value**.
- Changes do not affect other variables.

Example:

```
let a = 10;
let b = a;
b = 20; // a remains 10
```

Reference Types

- Stored by **reference**.
- Changes affect all references to the object.

Example:

```
let obj1 = { x: 1 };
let obj2 = obj1;
obj2.x = 5; // obj1.x also becomes 5
```

Key Takeaways (Additional)

- **Hoisting behavior differs between** `var` **and** `let/const` and can cause runtime errors.
- **Truthy/falsy evaluation** simplifies conditional logic but must be used carefully.
- **forEach does not return a new array**, unlike `map` and `filter`.
- **Async/await improves readability** over chained `.then()` calls.

- **Error handling is essential** for stable applications.
- **JSON is the standard data format for APIs**.
- **Understanding reference vs value types prevents unintended mutations**.