Introduction - JavaScript

- JavaScript is a high-level, dynamic, interpreted programming language that is well-suited to object-oriented and functional programming styles.
- JavaScript's variables are untyped. Its syntax is loosely based on Java, but the languages are otherwise unrelated.
- The name "JavaScript" is quite misleading. Except for a superficial syntactic resemblance, JavaScript is completely different from the Java programming language.
- "JavaScript" is a trademark licensed from Sun Microsystems (now Oracle) used to describe Netscape's (now Mozilla's) implementation of the language.
- Netscape submitted the language for standardization to ECMA—the European Computer Manufacturer's Association—and because of trademark issues, the standardized version of the language was stuck with the awkward name "ECMAScript."
- The language is popularly known as JavaScript. But the actual name is "ECMAScript" and the abbreviation "ES" to refer to the language standard and to versions of that standard.

Introduction – Hello World!!

- The easiest way to try out a few lines of JavaScript is to open up the web developer tools in our web browser (with F12, Ctrl-Shift-I, or Command-Option-I) and select the Console tab.
- We can use the function console.log() to display text and other JavaScript values in your terminal window or in a browser's developer tools console.
- Example: console.log("Hello World!!")
- Example: alert("Hello World")

Lexical Structure

- The lexical structure of a programming language is the set of elementary rules that specifies how you write programs in that language.
- It is the lowest-level syntax of a language: it specifies what variable names look like, the delimiter characters for comments, and how one program statement is separated from the next.

JavaScript is a case-sensitive language

- Keywords, variables, function names, and other identifiers must always be typed with a consistent capitalization of letters.
- The while keyword, for example, must be typed “while,” not “While” or “WHILE.”
- Similarly, online, Online, OnLine, and ONLINE are four distinct variable names.
- JavaScript ignores spaces that appear between tokens in programs.
- For the most part, JavaScript also ignores line breaks.
- JavaScript also recognizes tabs, assorted ASCII control characters, and various Unicode space characters as whitespace.
- JavaScript recognizes newlines, carriage returns, and a carriage return/line feed sequence as line terminators.

Lexical Structure

Comments

- Any text between a // and the end of a line is treated as a comment and is ignored by JavaScript.
- Any text between the characters /* and */ is also treated as a comment; these comments may span multiple lines but may not be nested.
- Example: // This is a single-line comment.
- Example: /* This is also a comment */   // and here is another comment.

Literals

- A literal is a data value that appears directly in a program.
- Example: 12, 1.2, “hello world”, ‘Hi’, true, false

Identifiers & Reserved Words

- In JavaScript, identifiers are used to name constants, variables, properties, functions, and classes and to provide labels for certain loops in JavaScript code.
- A JavaScript identifier must begin with a letter, an underscore (_), or a dollar sign ($).
- JavaScript reserves certain identifiers for use by the language itself, known as “reserved words” and cannot be used as regular identifiers.

Lexical Structure

Unicode Character Set

- JavaScript programs are written using the Unicode character set, and we can use any Unicode characters in strings and comments.
- It is common to use only ASCII letters and digits in identifiers.
- This means that programmers can use mathematical symbols and words from non-English languages as constants and variables.
- Example: const π = 3.14;
- Some computer hardware and software cannot display, input, or correctly process the full set of Unicode characters.
- JavaScript defines escape sequences that allow us to write Unicode characters using only ASCII characters.
- These Unicode escapes begin with the characters \u and are either followed by exactly four hexadecimal digits (using uppercase or lowercase letters A–F) or by one to six hexadecimal digits enclosed within curly braces.
- Example: let café = 1; caf\u00e9 caf\u{E9} console.log("\u{1F600}");

Lexical Structure

Optional Semicolons

- JavaScript uses the semicolon (;) to separate statements from one another.
- In JavaScript, we can usually omit the semicolon between two statements if those statements are written on separate lines.
- Example: a = 3; b = 4;
- Example: a = 3

  b = 4
- JavaScript treats a line break as a semicolon if the next non-space character cannot be interpreted as a continuation of the current statement.
- Example: let a;

  a

  =

  3
- JavaScript interprets this code like this: let a; a = 3; console.log(a);

Lexical Structure

Optional Semicolon

- If a statement begins with (, [, /, +, or -, there is a chance that it could be interpreted as a continuation of the statement before.
- Some programmers like to put a defensive semicolon at the beginning of any such statement so that it will continue to work correctly.
- Example:
```
let y = x + f
(a+b).toString()
```
  JavaScript interprets the code like this: `let y = x + f(a+b).toString();`
- Example:
```
let x = 0
;[x,x+1,x+2].forEach(console.log)     // Defensive ; keeps this statement separate
```
- There are three exceptions to the general rule that JavaScript interprets line breaks as semicolons when it cannot parse the second line as a continuation of the statement on the first line.
- The first exception involves the return, throw, yield, break, and continue statements.
- These statements often stand alone, but they are sometimes followed by an identifier or expression. If a line break appears after any of these words (before any other tokens), JavaScript will always interpret that line break as a semicolon.

Lexical Structure

Optional Semicolon

- Example: return

  true;

  JavaScript assumes: return; true;

  However, we probably meant: return true;
- This means that we must not insert a line break between return, break, or continue and the expression that follows the keyword. If we do insert a line break, our code is likely to fail in a nonobvious way that is difficult to debug.
- The second exception involves the ++ and −− operators. If we want to use either of these operators as postfix operators, they must appear on the same line as the expression they apply to.
- The third exception involves functions defined using concise “arrow” syntax: the => arrow itself must appear on the same line as the parameter list.

Types, Values & Variables

Types

- JavaScript types can be divided into two categories: **primitive types** and **object types**.
- JavaScript’s primitive types include numbers, strings of text (known as strings), and Boolean truth values (known as Booleans).
- The special JavaScript values null and undefined are primitive values, but they are not numbers, strings, or Booleans.
- Each value is typically considered to be the sole member of its own special type.
- ES6 adds a new special purpose type, known as Symbol, that enables the definition of language extensions without harming backward compatibility.
- Any JavaScript value that is not a number, a string, a Boolean, a symbol, null, or undefined is an object.
- An object (that is, a member of the type object) is a collection of properties where each property has a name and a value (either a primitive value or another object).
- Example: array, Set, Map, etc.

Types, Values & Variables

- The JavaScript interpreter performs automatic garbage collection for memory management.
- When a value is no longer reachable—when a program no longer has any way to refer to it—the interpreter knows it can never be used again and automatically reclaims the memory it was occupying.
- JavaScript supports an object-oriented programming style.
- This means that rather than having globally defined functions to operate on values of various types, the types themselves define methods for working with values.
- To sort the elements of an array a, for example, we don't pass a to a sort() function. Instead, we invoke the sort() method of a:

Example: a.sort();

Types, Values & Variables

- JavaScript’s object types are mutable and its primitive types are immutable.
- JavaScript liberally converts values from one type to another.
- Constants and variables allow us to use names to refer to values in our programs.
- Constants are declared with const and variables are declared with let.
- JavaScript constants and variables are untyped.

Numbers

- JavaScript's primary numeric type, Number, is used to represent integers and to approximate real numbers.
- JavaScript represents numbers using the 64-bit floating-point format defined by the IEEE 754 standard.
- When a number appears directly in a JavaScript program, it's called a numeric literal.
- Example: 0, 3, 10000
- In addition to base-10 integer literals, JavaScript recognizes hexadecimal (base-16) values (begins with 0x or 0X), binary (begins with 0b or 0B) and octal (begins with 0o or 0O).
- Floating-point literals can have a decimal point.
- Example: 3.14, 6.02e23 /* 6.02×10^{23} */, 1.4738223E-32 /* $1.4738223 \times 10^{-32}$ */
- We can use underscores within numeric literals to break long literals up into chunks that are easier to read:
- Example: let x = 1_00_00_235; let y = 0.123_456_789; let z = 0x89_AB_CD_EF;

Arithmetic in JavaScript

- JavaScript provides + for addition, - for subtraction, * for multiplication, / for division, and % for modulo. ES2016 adds ** for exponentiation.
- JavaScript supports more complex mathematical operations through a set of functions and constants defined as properties of the Math object.
- Example: Math.pow(2,3); Math.random(); Math.log(); Math.log2(); etc.
- Arithmetic in JavaScript does not raise errors in cases of overflow, underflow, or division by zero.
- When the result of a numeric operation is larger than the largest representable number (overflow), the result is a special infinity value, Infinity.
- when the absolute value of a negative value becomes larger than the absolute value of the largest representable negative number, the result is negative infinity, - Infinity.

Arithmetic in JavaScript

- There is one exception, however: zero divided by zero does not have a well-defined value, and the result of this operation is the special not-a-number value, NaN.
- NaN also arises if we attempt to divide infinity by infinity, take the square root of a negative number, or use arithmetic operators with non-numeric operands that cannot be converted to numbers.
- Infinity and NaN are available as properties of the Number object.
- Example: Number.POSITIVE_INFINITY; Number.NEGATIVE_INFINITY;

  Number.MAX_VALUE; Number.NaN; 0*'l‘; Infinity/Infinity;
- The not-a-number value has one unusual feature in JavaScript: it does not compare equal to any other value, including itself.
- This means that we can’t write x === NaN to determine whether the value of a variable x is NaN. Instead, we must write x != x or Number.isNaN(x).
- Those expressions will be true if, and only if, x has the same value as the global constant NaN.

Arithmetic in JavaScript

- The negative zero value is also somewhat unusual.
- It compares equal (even using JavaScript's strict equality test) to positive zero, which means that the two values are almost indistinguishable, except when used as a divisor.
- Example: let zero = 0; let negz = -0; zero === negz; 1/zero === 1/negz;

Dates and Times

- JavaScript Dates are objects, but they also have a numeric representation as a timestamp that specifies the number of elapsed milliseconds since January 1, 1970.
- Example: let timestamp = Date.now(); let now = new Date(); let ms = now.getTime();

Text

- A string is an immutable ordered sequence of 16-bit values, each of which typically represents a Unicode character.
- JavaScript's strings (and its arrays) use zero based indexing: the first 16-bit value is at position 0, the second at position 1, and so on.
- The empty string is the string of length 0.
- Example: let x ='a'; x.length;

String Literals

- To include a string in a JavaScript program, simply enclose the characters of the string within a matched pair of single or double quotes or backticks (' or " or `).
- Example: ‘abc’, “abc”, `abc` , ‘”abc”’,`’abc’` , 'two\nlines‘, “ab\”c\””
- If we use + operator on strings, it joins them by appending the second to the first.
- Example: let msg = "Hello, " + "world"; name = “Rajesh”

  let greeting = "Welcome to my blog," + " " + name;
- Different properties of String.

Template Literals

- String literals can be delimited with backticks.
- These template literals can include arbitrary JavaScript expressions. The final value of a string literal in backticks is computed by evaluating any included expressions, converting the values of those expressions to strings and combining those computed strings with the literal characters within the backticks.

String Literals

Template Literals

- Example: let name = "Bill";

  let greeting = `Hello ${ name }.`;
- Everything between the ${ and the matching } is interpreted as a JavaScript expression. Everything outside the curly braces is normal string literal text.
- The expression inside the braces is evaluated and then converted to a string and inserted into the template, replacing the dollar sign, the curly braces, and everything in between them.

Type Conversions

- JavaScript is very flexible about the types of values it requires.
- If JavaScript wants a string, it will convert whatever value we give it to a string. If JavaScript wants a number, it will try to convert the value we give it to a number.
- Example: 10 + " objects“

  "7" * "4“

  let n = 1 - "x"; n + " objects“;

Conversions and Equality

- JavaScript has two operators that test whether two values are equal.
- The “strict equality operator,” ===, does not consider its operands to be equal if they are not of the same type.
- It also defines the == operator with a flexible definition of equality.
- Example: "0" == 0; 0 == false; "0" == false;
- But sometimes we need to perform an explicit conversion.
- The simplest way to perform an explicit type conversion is to use the Boolean(), Number(), and String() functions.
- Example: Number("3"); String(false); Boolean([]);
- Any value other than null or undefined has a toString() method, and the result of this method is usually the same as that returned by the String() function.

Expressions and Operators

Primary Expressions

- The simplest expressions, known as primary expressions, are those that stand alone—they do not include any simpler expressions.
- Primary expressions in JavaScript are constant or literal values, certain language keywords, and variable references.

Array Initializers

- Object and array initializers are expressions whose value is a newly created object or array.
- These initializer expressions are sometimes called object literals and array literals.
- They are not primary expressions, because they include a number of subexpressions that specify property and element values.
- An array initializer is a comma-separated list of expressions contained within square brackets.
- Example: [1+2,3+4]; [];
- Example: let matrix = [[1,2,3], [4,5,6], [7,8,9]];
- Undefined elements can be included in an array literal by simply omitting a value between commas.
- Example: let sparseArray = [1,,,,5];

Expressions and Operators

Object Initializers

- Object initializer expressions are like array initializer expressions, but the square brackets are replaced by curly brackets.
- Each subexpression is prefixed with a property name and a colon ":"
- Example: let p = { x: 2.3, y: -1.2 }; let q = {}; q.x = 2.3; q.y = -1.2;
- Object literals can be nested.
- Example: let rectangle = {

```
        upperLeft: { x: 2, y: 2 },
        lowerRight: { x: 4, y: 5 }
    };
```

Property Access Expressions

- A property access expression evaluates to the value of an object property or an array element.
- JavaScript defines two syntaxes for property access.
- expression . identifier & expression [expression]
- The first style of property access is an expression followed by a period and an identifier.
- The second style of property access follows the first expression (the object or array) with another expression in square brackets.
- Example: let o = {x: 1, y: {z: 3}}; let a = [0, 4, [5, 6]];

  o.x; o.y.z; o["x"]; a[1]; a[2]["1"]; a[0].x

null & undefined

- null is a language keyword that evaluates to a special value that is usually used to indicate the absence of a value.
- Using the typeof operator on null returns the string “object”, indicating that null can be thought of as a special object value that indicates “no object”.
- null is typically regarded as the sole member of its own type, and it can be used to indicate “no value” for numbers and strings as well as objects.
- JavaScript also has a second value that indicates absence of value (viz.) the undefined value represents a deeper kind of absence.
- It is the value of variables that have not been initialized and the value you get when you query the value of an object property or array element that does not exist.

null & undefined

- The undefined value is also the return value of functions that do not explicitly return a value and the value of function parameters for which no argument is passed.
- undefined is a predefined global constant that is initialized to the undefined value.
- If we apply the typeof operator to the undefined value, it returns “undefined”, indicating that this value is the sole member of a special type.
- Since both indicate absence of value and can often be used interchangeably.
- The equality operator == considers them to be equal.

Conditional Property Access

- ES2020 adds two new kinds of property access expressions:
  - expression ?. identifier
  - expression ?.[expression]
- We can use ?. and ?.[] syntax to guard against errors of this type.
- We get a TypeError if the expression on the left evaluates to null or undefined.
- We can use ?. and ?.[] syntax to guard against errors of this type.
- Consider the expression a?.b. If a is null or undefined, then the expression evaluates to undefined without any attempt to access the property b.
- If a is some other value, then a?.b evaluates to whatever a.b would evaluate to.
- And if a does not have a property named b, then the value will again be undefined.
- This form of property access expression is sometimes called "optional chaining" because it also works for longer "chained" property access.
- Example: let a = { b: null }; a.b?.c.d
- What is the output?: (a.b?.c).d ; let f = null ; f(3) ; f?.(3)

Arithmetic Expressions

- The ** operator has higher precedence than *, /, and % (which in turn have higher precedence than + and -).
- Example: 2**2**3; -3**2 ; (-3)**2
- The / operator divides its first operand by its second.
- In JavaScript, however, all numbers are floating-point, so all division operations have floating-point results: 5/2 evaluates to 2.5, not 2.
- While the modulo operator is typically used with integer operands, it also works for floating-point values. For example, 6.5 % 2.1 evaluates to 0.2.

Arithmetic Expressions

+ Operator

- If either of its operand values is an object, it converts it to a primitive using the object-to-primitive algorithm.
- if either operand is a string, the other is converted to a string and concatenation is performed.
- Otherwise, both operands are converted to numbers
- Example: 1 + 2; “1” + “2”; 1 + a; true + true; 1 + a[]; 1 + null; 1 + undefined;
- It is important to note that when the + operator is used with strings and numbers, it may not be associative.
- Example: 1 + 2 + “hello”; 1 + (2 + “hello”)

Unary Operators

Bitwise Operators

Relational Expressions

- Strict Equality
- Equality Operator
- Comparison Operators
- In Operator
  - It evaluates to true if the left-side value is the name of a property of the right-side object.
  - Example: let point = {x: 1, y: 1}; "x" in point; let data = [7,8,9]; "0" in data;
- Instanceof Operator
  - The instanceof operator expects a left-side operand that is an object and a right-side operand that identifies a class of objects.
  - Example: let d = new Date(); d instanceof Date; d instanceof Object; let a = [1, 2, 3];

Types, Values & Variables

eval()

- eval() expects one argument.
- If we pass any value other than a string, it simply returns that value.
- If you pass a string, it attempts to parse the string as JavaScript code, throwing a SyntaxError if it fails.
- If it successfully parses the string, then it evaluates the code and returns the value of the last expression or statement in the string or undefined if the last expression or statement had no value.
- Example: eval("var x = 3"); let y = 3; eval("y");

First-Defined(??)

- The first-defined operator ?? evaluates to its first defined operand: if its left operand is not null and not undefined, it returns that value.
- Otherwise, it returns the value of the right operand.
- Example: let y = 3; y ?? 4; let z; z ?? 9;

The delete Operator

- delete is a unary operator that attempts to delete the object property or array element specified as its operand.
- Example:
  - let o = { x: 1, y: 2};
  - delete o.x;
  - let a = [1,2,3,4];
  - delete a[2];
- In strict mode, delete raises a SyntaxError if its operand is an unqualified identifier such as a variable, function, or function parameter: it only works when the operand is a property access expression.
- Strict mode also specifies that delete raises a TypeError if asked to delete any non-configurable (i.e., nondeleteable) property.
- Outside of strict mode, no exception occurs in these cases, and delete simply returns false to indicate that the operand could not be deleted.

Compound and Empty Statements

- A statement block combines multiple statements into a single compound statement. A statement block is simply a sequence of statements enclosed within curly braces.
- A compound statement allows you to use multiple statements where JavaScript syntax expects a single statement.
- The empty statement is the opposite: it allows you to include no statements where one is expected.

Conditionals

- General Syntax

```
if (expression)
        statement1
else if (expression)
        statement 2
else
        statement3
```

- What is the Ouput?

```
if (i === j) {
        if (j === k)
                console.log("i equals k");
else
        console.log("i doesn't equal j"); // OOPS!
}
```

Conditionals & Loops

- Switch expressions
- While loop
- Do-while loop
- For loop
- for/of loop
  - ES6 defines a new loop statement: for/of.
  - The for/of loop works with iterable objects.
  - Example:

```
let data = [1, 2, 3, 4, 5, 6, 7, 8, 9], sum = 0;
for(let element of data) {
        sum += element;
}
sum
```

Loops

- Example:

```
let o = { x: 1, y: 2, z: 3 };
let keys = "";
for(let k of Object.keys(o)) {
        keys += k;
}
keys
```

- Example:

```
let sum = 0;
for(let v of Object.values(o)) {
        sum += v;
}
sum
```