JavaScript-2

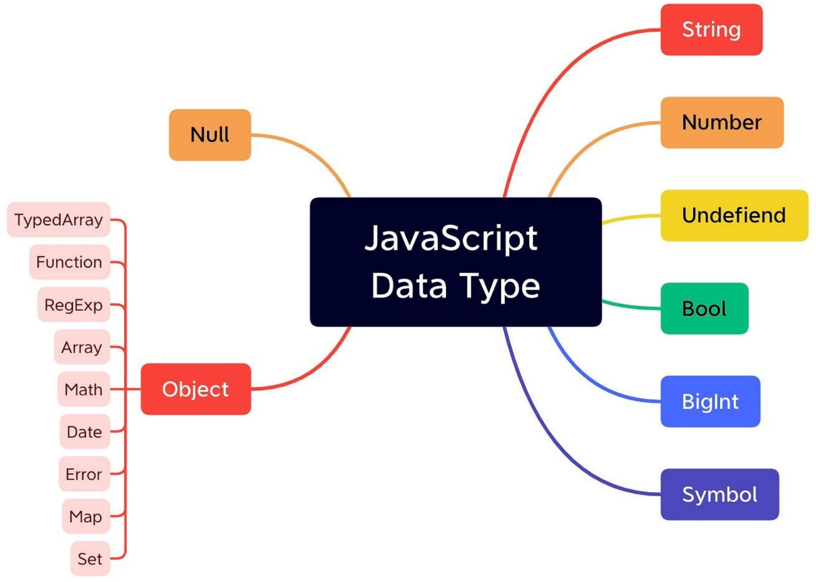
Nesli Erdogmus

neslierdogmus@iyte.edu.tr

JavaScript data types

- JavaScript has 8 data types:
  1. String
  2. Number
  3. BigInt
  4. Boolean
  5. Undefined
  6. Null
  7. Symbol
  8. Object

  } Primitive values (immutable)
- You can use the `typeof` operator to find the data type of a JavaScript variable.



https://siwonlog.hashnode.dev/javascript-data-type-and-memory

These slides are composed using the following online sources: https://developer.mozilla.org/en-US/docs/Learn/CSS https://www.w3schools.com/css/default.asp

JavaScript data types

- **BigInt:** These variables are used to store big integer values that are too big to be represented by a normal JavaScript Number.
    - It was introduced in ES11.
- **Undefined:** A variable without a value, has the value undefined. The type is also undefined.
- **Null:** This type is inhabited by exactly one value: null, which represents the intentional absence of any object value. (`typeof null === "object";`)
- **Number** has some special values:
    - **NaN:** NaN ("Not a Number") is a special kind of number value that's typically encountered when the result of an arithmetic operation cannot be expressed as a number. It is also the only value in JavaScript that is not equal to itself.
    - **+Infinity, -Infinity:** A number reaches +/-Infinity when it exceeds the upper/lower limit for a number.

JavaScript data types

- **Symbol:** The purpose of symbols is to create unique property keys that are guaranteed not to clash with keys from other code.
  - It was introduced in ES6.
  - It represents a unique "hidden" identifier that no other code can accidentally access.

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

let id = Symbol('id');
person[id] = 140353;
// Now person[id] = 140353
// but person.id is still undefined
```

JavaScript objects

- In JavaScript, objects are the only mutable values.
- They can be seen as a collection of properties.
  - Object properties are equivalent to key-value pairs.
    - Property keys are either strings or symbols. Property values can be values of any type.
- JavaScript has many standard, built-in objects such as:
  - Fundamental objects: Object, Function, Boolean, Symbol
  - Error objects: Error, RangeError, TypeError, URIError, …
  - Numbers and dates: Number, BigInt, Math, Date
  - Text processing: String, RegExp
  - Indexed collections: Array, Int8Array, Uint32Array, BigInt64Array, Float32Array, …
  - Keyed collections: Map, Set, WeakMap, WeakSet
  - …

JavaScript objects

- Syntax:
  - JavaScript objects are written with curly braces {}.
  - Object properties are written as name:value pairs, separated by commas.
  - You can access object properties in two ways:
    - `objectName.propertyName`
    - `objectName["propertyName"]`

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

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

```
person.lastName;
```

```
person["lastName"];
```

JavaScript objects

- In the previous example, the **`person`** object was created using an object literal. There are other ways to create new objects:
  - **`new`** operator lets developers create an instance of a user-defined object type or of one of the built-in object types that has a constructor function.

```
const person = new Object();
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

  - For readability, simplicity and execution speed, use the object literal method.

JavaScript objects

- JavaScript Objects are mutable: they are addressed by reference, not by value.
- If person is an object, the following statement will not create a copy of person:

```
const x = person;  // Will not create a copy of person.
```

  - The object x is not a copy of person. It is person. Both x and person are the same object.
  - Any changes to x will also change person, because x and person are the same object.

JavaScript objects

- In JavaScript, the **`this`** keyword refers to an object.
  - Which object depends on how this is being invoked (used or called).
    - In an object method, `this` refers to the **object**.
    - Alone, `this` refers to the **global object**.
    - In a function, `this` refers to the **global object**.
    - In a function, in strict mode, `this` is `undefined`.
    - In an event, `this` refers to the **element** that received the event.
    - Methods like `call()`, `apply()`, `and bind()` can refer `this` to **any object**.

JavaScript Object Notation (JSON)

- JSON is a lightweight data-interchange format, derived from JavaScript, but used by many programming languages.
- JSON is built on two structures:
  - A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
  - An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

https://www.json.org/json-en.html

JavaScript Object Notation (JSON)

- In JavaScript, the JSON namespace object contains static methods for parsing values from and converting values to JavaScript Object Notation (JSON).

```
{
  "browsers": {
    "firefox": {
      "name": "Firefox",
      "pref_url": "about:config",
      "releases": {
        "1": {
          "release_date": "2004-11-09",
          "status": "retired",
          "engine": "Gecko",
          "engine_version": "1.7"
        }
      }
    }
  }
}
```

```
const jsonText = `{
  "browsers": {
    "firefox": {
      "name": "Firefox",
      "pref_url": "about:config",
      "releases": {
        "1": {
          "release_date": "2004-11-09",
          "status": "retired",
          "engine": "Gecko",
          "engine_version": "1.7"
        }
      }
    }
  }
}`;

console.log(JSON.parse(jsonText));
```

JavaScript functions

- Functions are one of the fundamental building blocks in JavaScript.
- In JavaScript, functions are first-class objects, because they can be passed to other functions, returned from functions, and assigned to variables and properties.
- Functions can also have properties and methods just like any other object.
    - In JavaScript, every function is actually a Function object.
    - Callable values cause typeof to return "function" instead of "object".

JavaScript functions

- Defining functions
  - A function definition (also called a function declaration, or function statement) consists of the `function` keyword, followed by:
    - The name of the function.
    - A list of parameters to the function, enclosed in parentheses and separated by commas.
    - The JavaScript statements that define the function, enclosed in curly brackets, { /* … */ }.

```
// Function to compute the product of p1 and p2
function myFunction(p1, p2) {
  return p1 * p2;
}
```

    - The function **`myFunction`** takes two parameters, called **`p1`** and **`p2`**. The function consists of one statement that says to return the multiplication of the function (that is, **`myFunction`**) parameters.

JavaScript functions

- Defining functions
  - Functions can also be created by a function expression. Such a function can be anonymous; it does not have to have a name.

```
// Function to compute the product of p1 and p2
const myFunction = function(p1, p2) {
  return p1 * p2;
}
```

  - However, a name can be provided with a function expression.
    - Providing a name allows the function to refer to itself, and also makes it easier to identify the function in a debugger's stack traces.

```
// Function to compute the product of p1 and p2
const myFunction = function product(p1, p2) {
  return p1 * p2;
}
```

JavaScript functions

- Function parameters
  - JavaScript function definitions do not specify data types for parameters.
    - Type of the passed arguments is not checked!
    - Number of the passed arguments is not checked!
  - Parameters are essentially passed to functions **by value** — so if the code within the body of a function assigns a completely new value to a parameter that was passed to the function, the change is not reflected globally or in the code which called that function.
  - When you pass an object or an array as a parameter, the value is the object reference. So, objects will behave like they are passed by reference: if the function changes the object's properties, that **change is visible outside the function**.

JavaScript functions

- Function parameters

```
function myFunc(theObject) {
  theObject.make = "Toyota";
}

const mycar = {
  make: "Honda",
  model: "Accord",
  year: 1998,
};

// x gets the value "Honda"
const x = mycar.make;

// the make property is changed by the function
myFunc(mycar);
// y gets the value "Toyota"
const y = mycar.make;
```

JavaScript functions

- Function scope
    - Variables defined inside a function cannot be accessed from anywhere outside the function, because the variable is defined only in the scope of the function.
    - A function can access all variables and functions defined inside the scope in which it is defined.

```
// The following variables are defined in the global scope
const num1 = 20;
const num2 = 3;
const name = "Chamakh";

// This function is defined in the global scope
function multiply() {
  return num1 * num2;
}
multiply(); // Returns 60

// A nested function example
function getScore() {
  const num1 = 2;
  const num2 = 3;

  function add() {
    return `${name} scored ${num1 + num2}`;
  }

  return add();
}
getScore();  // Returns "Chamakh scored 5"
```

JavaScript functions

- Arrow functions
  - An arrow function expression (also called a fat arrow to distinguish from a hypothetical -> syntax in future JavaScript) has a shorter syntax compared to function expressions.
  - It does not have its own `this`, `arguments`, `super`, or `new.target`.
  - Arrow functions are always anonymous.

```
const a = ["Hydrogen", "Helium", "Lithium", "Beryllium"];

const a2 = a.map(function (s) {
  return s.length;
});

console.log(a2); // [8, 6, 7, 9]

const a3 = a.map((s) => s.length);

console.log(a3); // [8, 6, 7, 9]
```

JavaScript functions

- JavaScript has several top-level, built-in functions:
  - eval()
  - isFinite()
  - isNaN()
  - parseFloat()
  - parseInt()
  - decodeURI()
  - decodeURIComponent()
  - encodeURI()
  - encodeURIComponent()
  - escape()
  - unescape()

JavaScript classes

- JavaScript classes were introduced with ES6.
- They are templates for JavaScript objects.
- Classes are in fact "special functions".
  - Just as you can define function expressions and function declarations, a class can be defined in two ways: a class expression or a class declaration.
  - The body of a class is the part that is in curly brackets {} where you define class members, such as methods or constructor.

```
// Declaration
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

```
// Expression; the class is anonymous
but assigned to a variable
const Rectangle = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

```
// Expression; the class has its own
name
const Rectangle = class Rectangle2 {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

JavaScript classes

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Getter
  get area() {
    return this.calcArea();
  }
  // Method
  calcArea() {
    return this.height * this.width;
  }
  *getSides() {
    yield this.height;
    yield this.width;
    yield this.height;
    yield this.width;
  }
}

const square = new Rectangle(10, 10);

console.log(square.area); // 100
console.log([...square.getSides()]); // [10, 10, 10, 10]
```

Asynchronous JavaScript

- Asynchronous programming is a technique that enables your program to start a potentially long-running task and still be able to be responsive to other events while that task runs, rather than having to wait until that task has finished.
- Many functions provided by browsers, especially the most interesting ones, can potentially take a long time, and therefore, are asynchronous:
  - Making HTTP requests using `fetch()`
  - Accessing a user's camera or microphone using `getUserMedia()`
  - Asking a user to select files using `showOpenFilePicker()`

Asynchronous JavaScript

- Event handlers
    - Event handlers are a form of asynchronous programming: you provide a function (the event handler) that will be called, not right away, but whenever the event happens.
    - An event handler is a particular type of callback.
        - A callback is a function that's passed into another function, with the expectation that the callback will be called at the appropriate time.
        - Callback-based code can get hard to understand when the callback itself has to call functions that accept a callback. (When you need to perform a series of asynchronous functions.)
        - With **`promise`**s, we accomplish this by creating a promise chain.

Asynchronous JavaScript

```javascript
function doStep1(init) {
  return init + 1;
}

function doStep2(init) {
  return init + 2;
}

function doStep3(init) {
  return init + 3;
}

function doOperation() {
  let result = 0;
  result = doStep1(result);
  result = doStep2(result);
  result = doStep3(result);
  console.log(`result: ${result}`);
}

doOperation();
```

```javascript
function doStep1(init, callback) {
  const result = init + 1;
  callback(result);
}

function doStep2(init, callback) {
  const result = init + 2;
  callback(result);
}

function doStep3(init, callback) {
  const result = init + 3;
  callback(result);
}

function doOperation() {
  doStep1(0, (result1) => {
    doStep2(result1, (result2) => {
      doStep3(result2, (result3) => {
        console.log(`result: ${result3}`);
      });
    });
  });
}

doOperation();
```

Asynchronous JavaScript

- **`Promise`**
  - A Promise is an object returned by an asynchronous function, which represents the current state of the operation.
  - At the time the promise is returned to the caller, the operation often isn't finished, but the promise object provides methods to handle the eventual success or failure of the operation.
  - We can create a promise chain using the **`then()`** function which returns a new promise, different from the original:

```
const promise = doSomething();
const promise2 = promise.then(successCallback, failureCallback);
```

Asynchronous JavaScript

```
doSomething(function (result) {
  doSomethingElse(result, function (newResult) {
    doThirdThing(newResult, function (finalResult) {
      console.log(`Got the final result: ${finalResult}`);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

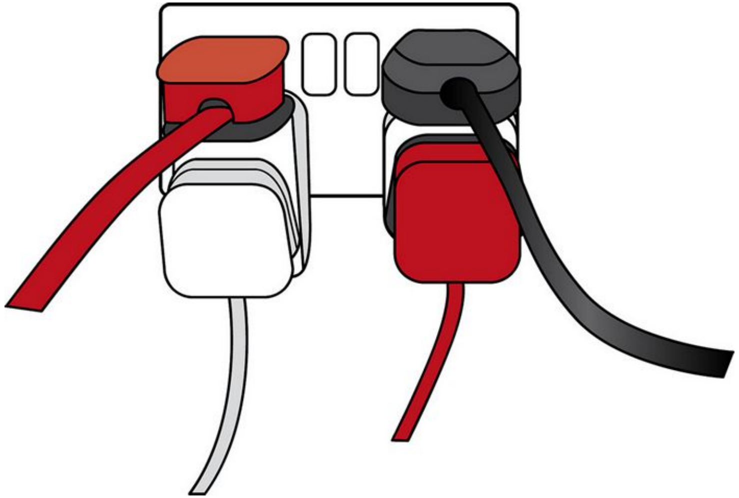
```
doSomething()
  .then(function (result) {
    return doSomethingElse(result);
  })
  .then(function (newResult) {
    return doThirdThing(newResult);
  })
  .then(function (finalResult) {
    console.log(`Got the final result: ${finalResult}`);
  })
  .catch(failureCallback);
```

The arguments to **`then`** are optional, and **`catch(failureCallback)`** is short for **`then(null, failureCallback)`**.

```
doSomething()
  .then((result) => doSomethingElse(result))
  .then((newResult) => doThirdThing(newResult))
  .then((finalResult) => {
    console.log(`Got the final result: ${finalResult}`);
  })
  .catch(failureCallback);
```

Client-side web APIs

- Application Programming Interfaces (APIs) are constructs made available in programming languages to allow developers to create complex functionality more easily.



Client-side web APIs

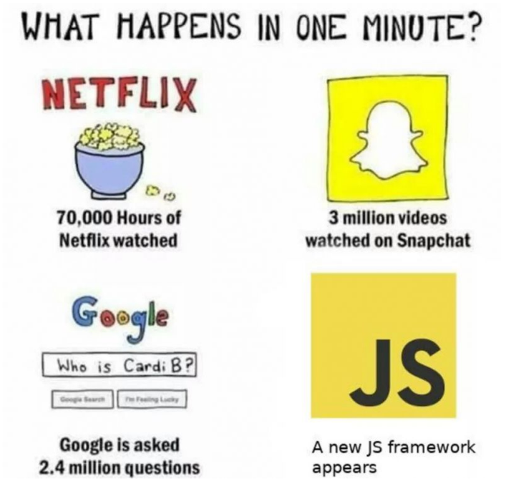
- When writing client-side JavaScript for web sites or applications, you will encounter different APIs:
  - **APIs for manipulating documents** loaded into the browser. The most obvious example is the **DOM (Document Object Model) API**, which allows you to manipulate HTML and CSS — creating, removing and changing HTML, dynamically applying new styles to your page, etc.
  - **APIs that fetch data from the server** to update small sections of a webpage on their own are very commonly used. The main API used for this is the **Fetch API**, although older code might still use the XMLHttpRequest API.
  - **Client-side storage APIs** enable you to store data on the client-side, so you can create an app that will save its state between page loads, and perhaps even work when the device is offline. (**Web Storage API**, **IndexedDB API**)
  - …

Client-side JavaScript frameworks

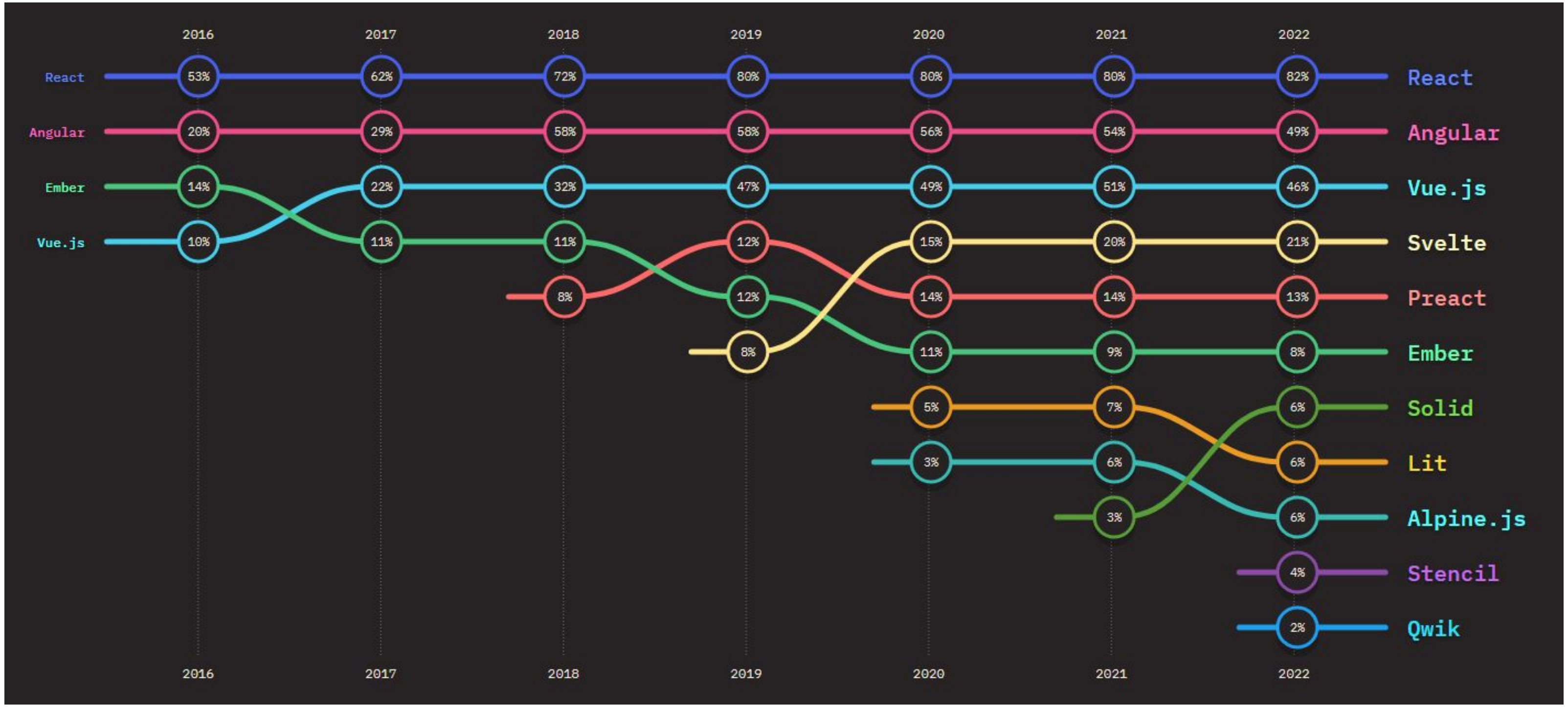
- Today, JavaScript is an essential part of the web, used on 95% of all websites, and the web is an essential part of modern life.
- The web allows us to do things that used to be possible only in native applications installed on our computers. These modern, complex, interactive websites are often referred to as **web applications**.
- A framework is a library that offers opinions about how software gets built.
  - Predictable
    - Scalable
    - Maintainable
  - Homogenous

Client-side JavaScript frameworks

- JavaScript frameworks are an essential part of modern front-end web development, providing developers with tried and tested tools for building scalable, interactive web applications.
- Where to begin
  - There are so many frameworks to choose from.
  - New ones appear all the time.
  - They mostly work in a similar way but do some things differently.


WHAT HAPPENS IN ONE MINUTE?
NETFLIX
70,000 Hours of Netflix watched
3 million videos watched on Snapchat
Google
Who is Cardi B?
Google is asked 2.4 million questions
JS
A new JS framework appears

Client-side JavaScript frameworks

- 

React

- Facebook released React in 2013.
- Technically, React itself is not a framework; it's a library for rendering UI components.
- It is used in combination with other libraries to make applications:
    - React and React Native enable developers to make mobile applications.
    - React and ReactDOM enable developers to make web applications.
    - …
- React extends JavaScript with HTML-like syntax, known as **JSX**.

```
const heading = <h1>Mozilla Developer Network</h1>;
```