JavaScript Lecture 1

JavaScript is a versatile and powerful programming language that has become an essential part of web development.

> 💡 It enables developers to create interactive and dynamic web pages, enhancing both user experience and functionality.

Unlike HTML and CSS, which primarily focus on structure and styling, JavaScript enables complex logic and behavior to be implemented directly in the browser.

- ▼ Bibloteque:
  - ▼ Basics：

    What is Javascript?

    History and Versions （ES5（2009）, ES2015/ES6, TypeScript, JSX）

    Development Tools and Installation

    VS Code and NodeJS

    First Application (console.log)

    Lexical Structure

Variables and Data Types

Operators and Expressions

Statements

▼ Object-oriented Programming:

What is JavaScript?

In the era before ES5, JavaScript is normally defined an interpreted (or scripting) programming language, but was not recognized as Object-oriented Programming(OOP) language.

- primarily used for building web applications
- “just-in-time compiling”
- Used to be considered ‘Client Side’ technology
- Embedded in your HTML page
- Interpreted by the Web browser
- Now, it is also widely used on ‘Server Side’
- Via NodeJS
- With a large ecosystem

JavaScript is the only programming language, beside HTML and CSS that has been privileged to run all over browsers.

A little question here !

What is programming?

Programming is a set of instructions to tell the computer what to do.

These instructions are just words in plain English.

The type of words you see are known as the syntax.

Most languages use the same types of comments.

What is OOP? What are the characteristics?

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into objects—instances of classes. It's built on the idea of objects containing both data and code, enabling a more structured and modular approach to software development. The key characteristics of OOP are:

Encapsulation, Inheritance, Polymorphism, and Abstraction

These principles allow objects to interact and collaborate to create applications

> Encapsulation is a fundamental principle of object-oriented programming that bundles data and the methods that operate on that data within a single unit or object. It restricts direct access to some of an object's components, which is a means of preventing accidental interference and misuse of the methods and data.

```
class BankAccount {
  #balance = 0;  // Private field

  constructor(initialBalance) {
    this.#balance = initialBalance;
  }

  deposit(amount) {
    if (amount > 0) {
      this.#balance += amount;
      return true;
    }
    return false;
  }

  withdraw(amount) {
    if (amount > 0 && amount <= this.#balance) {
```

```
      this.#balance -= amount;
      return true;
    }
    return false;
  }

  getBalance() {
    return this.#balance;
  }
}

const account = new BankAccount(1000);
console.log(account.getBalance());  // 1000
account.deposit(500);
console.log(account.getBalance());  // 1500
account.withdraw(200);
console.log(account.getBalance());  // 1300
// console.log(account.#balance);  // This would cause an error
```

In this example, the BankAccount class encapsulates the balance data, making it private and only accessible through public methods. This prevents direct manipulation of the balance, ensuring data integrity and demonstrating the principle of encapsulation.

1. Encapsulation

Encapsulation is the concept of bundling the data (properties) and the methods (functions) that operate on the data into a single unit, usually a class, and restricting access to some of the object's components. This is usually done using private properties and methods.

Example:

```
 javascriptCopy code
class Person {
    // Private field
    #name;

    constructor(name, age) {
        this.#name = name;
        this.age = age;
    }

    // Public method
    getName() {
        return this.#name;
```

```
    }

    setName(newName) {
        this.#name = newName;
    }

    // Public method
    getAge() {
        return this.age;
    }

    setAge(newAge) {
        if (newAge > 0) {
            this.age = newAge;
        }
    }
}

const person = new Person('John', 30);
console.log(person.getName()); // John
person.setName('Doe');
console.log(person.getName()); // Doe

// Direct access to #name will fail
console.log(person.#name); // SyntaxError: Private field '#name' must be declared in an enclosing class
```

In this example, `#name` is a private field that cannot be accessed directly from outside the class. We use public methods `getName()` and `setName()` to interact with it.

2. Inheritance

Inheritance is a mechanism where one class (child) can inherit properties and methods from another class (parent).

Example:

```
javascriptCopy code
class Animal {
    constructor(name) {
        this.name = name;
    }

    speak() {
        console.log(`${this.name} makes a sound.`);
    }
}

class Dog extends Animal {
    speak() {
        console.log(`${this.name} barks.`);
    }
```

```
}

const dog = new Dog('Rex');
dog.speak(); // Rex barks.

use strict;
public/ private
```

Here, `Dog` extends `Animal`, inheriting the `name` property and `speak` method. However, `Dog` overrides the `speak` method to provide its own implementation.

3. Polymorphism

Polymorphism allows objects to be treated as instances of their parent class rather than their actual class. It also allows methods to do different things based on the object's class.

Example:

```
javascriptCopy code
class Shape {
    draw() {
        console.log('Drawing a shape');
    }
}

class Circle extends Shape {
    draw() {
        console.log('Drawing a circle');
    }
}

class Square extends Shape {
    draw() {
        console.log('Drawing a square');
    }
}

const shapes = [new Circle(), new Square()];

shapes.forEach(shape => {
    shape.draw();
});

// Output:
// Drawing a circle
// Drawing a square
```

In this example, the `draw` method is defined in the `Shape` class and overridden in the `Circle` and `Square` classes. The `draw` method is called polymorphically, depending on the actual object type.

4. Abstraction

Abstraction involves hiding the complex implementation details and showing only the essential features of an object. In JavaScript, this can be achieved through abstract classes and methods. JavaScript doesn't have native support for abstract classes, but you can simulate it.

Example:

```
javascriptCopy code
class Shape {
    constructor() {
        if (new.target === Shape) {
            throw new Error('Cannot instantiate abstract class Shape directly');
        }
    }

    draw() {
        throw new Error('Method "draw()" must be implemented.');
    }
}

class Triangle extends Shape {
    draw() {
        console.log('Drawing a triangle');
    }
}

class Rectangle extends Shape {
    draw() {
        console.log('Drawing a rectangle');
    }
}

const shapes = [new Triangle(), new Rectangle()];

shapes.forEach(shape => {
    shape.draw();
});

// Output:
// Drawing a triangle
// Drawing a rectangle
```

In this example, `Shape` acts as an abstract class. It prevents direct instantiation and enforces that derived classes must implement the `draw` method.

These examples illustrate encapsulation, inheritance, polymorphism, and abstraction in JavaScript, demonstrating how these core OOP principles can be applied in the language.

Summary

- **Primitive Types**: String, Number, BigInt, Boolean, undefined, null, Symbol
- **Reference Types**: Object, Array, Function, Date, RegExp, Map, Set, WeakMap, WeakSet

Each type serves a specific purpose and has its own set of operations and properties. Understanding these types is fundamental to working effectively with JavaScript.

JavaScript has a variety of data types that can be categorized into primitive types and reference types. Here's a comprehensive overview of all the data types in JavaScript:

Primitive Data Types

Primitive data types are basic data types that are immutable (cannot be changed). They are stored directly in the variable.

1. **String**
   - Represents sequences of characters.
   - Example: `'Hello, world!'`, `"JavaScript"`, `` `Template Literal` ``
2. **Number**
   - Represents numeric values, both integers and floating-point numbers.
   - Example: `42`, `3.14`, `7`

3. **BigInt**
    - Represents integers with arbitrary precision (beyond the safe integer limit of `Number`).
    - Example: `1234567890123456789012345678901234567890n`
4. **Boolean**
    - Represents a logical value that can be either `true` or `false`.
    - Example: `true`, `false`
5. **undefined**
    - Represents a variable that has been declared but not yet assigned a value.
    - Example: `let x; console.log(x); // undefined`
6. **null**
    - Represents the intentional absence of any object value. It is a special value indicating "no value".
    - Example: `let y = null;`
7. **Symbol** (Introduced in ECMAScript 2015)
    - Represents a unique and immutable identifier.
    - Example: `Symbol('description')`
8. **Object** (Sometimes included with primitives)
    - Represents a collection of key-value pairs. While technically not a primitive, it is often used alongside primitives in a variety of ways.
    - Example: `{ key: 'value' }`

Reference Data Types

Reference data types refer to objects and are mutable (can be changed). They are stored as references to the actual data.

1. **Object**
    - The most generic type of reference. An object is a collection of key-value pairs.

- Example: `{ name: 'John', age: 30 }`

2. **Array**
   - A special type of object used for ordered collections of values.
   - Example: `[1, 2, 3, 4]`
3. **Function**
   - A type of object that represents a callable piece of code.
   - Example: `function greet() { return 'Hello!'; }`
4. **Date**
   - Represents dates and times.
   - Example: `new Date()`
5. **RegExp**
   - Represents regular expressions used for pattern matching in strings.
   - Example: `/^[a-z]+$/i`
6. **Map**
   - A collection of key-value pairs where keys and values can be of any type.
   - Example: `new Map([[1, 'one'], [2, 'two']])`
7. **Set**
   - A collection of unique values.
   - Example: `new Set([1, 2, 3, 3])`
8. **WeakMap**
   - A collection of key-value pairs where keys are objects and values can be of any type. WeakMap keys are held weakly, meaning they don't prevent garbage collection.
   - Example: `new WeakMap([[{}, 'value']])`
9. **WeakSet**
   - A collection of unique objects where object references are held weakly.

- Example: `new WeakSet([{}, {}])`

Special Cases

- **NaN (Not-a-Number)**
  - A special value representing an invalid number.
  - Example: `0 / 0`
- **Infinity and -Infinity**
  - Special values representing positive and negative infinity.
  - Example: `1 / 0` (Infinity), `1 / 0` (-Infinity)

Type Conversion

JavaScript also provides mechanisms to convert between types, such as using `String()`, `Number()`, and `Boolean()` functions.