Brendan Eich → Netscape Communications
↳ 1990 (Javascript)

JavaScript is a high-level programming language that is primarily used for creating interactive and dynamic web pages. It was developed by Brendan Eich at Netscape Communications in the mid-1990s and has since become one of the most widely used programming languages for both front-end and back-end web development.

Here are some key points about JavaScript:

1. **Client-Side Language:** JavaScript is primarily executed on the client-side, meaning it runs in the user's web browser. This allows for the creation of dynamic web content that can respond to user interactions without requiring constant communication with the server.

2. **Syntax and Features:** JavaScript has a C-like syntax and is an object-oriented language. It supports various programming paradigms, including procedural, functional, and object-oriented programming. JavaScript features dynamic typing, automatic memory management, and a rich set of built-in functions and objects.

3. **Web Development:** JavaScript is an integral part of web development. It enables developers to manipulate and modify web page elements, handle events, create animations, validate forms, and interact with APIs (Application Programming Interfaces) for fetching data from servers or integrating with external services.

4. **Frameworks and Libraries:** JavaScript has a vast ecosystem of frameworks and libraries that simplify web development. Some popular JavaScript frameworks include React.js, AngularJS, and Vue.js, which provide powerful tools and abstractions for building complex web applications.

5. **Node.js:** In addition to client-side development, JavaScript can also be used on the server-side with the help of Node.js. Node.js is a runtime environment that allows JavaScript to run outside the browser, enabling developers to build server-side applications, APIs, and real-time web services using JavaScript.

6. **Compatibility:** JavaScript is supported by all modern web browsers, making it a widely accessible language. However, browser compatibility issues can still arise due to differences in JavaScript implementations across browsers, requiring developers to handle them with appropriate techniques and libraries.

7. **Integration:** JavaScript can be easily integrated into HTML documents using `<script>` tags or included as separate external files. It can also be embedded in other programming languages or frameworks, allowing developers to extend the functionality of their applications.

8. **Frameworks for Mobile and Desktop:** JavaScript can be used to develop mobile applications using frameworks like React Native and Ionic. It can also be employed in desktop application development using frameworks such as Electron.

JavaScript is constantly evolving, with new features and improvements being introduced regularly. It has become a versatile and essential tool for web developers, enabling them to create dynamic and interactive experiences on the web.

JavaScript features:

1. **Dynamic Typing:** JavaScript is a dynamically typed language, which means you don't need to explicitly declare the data type of a variable. The type of a variable can change dynamically during runtime.

2. **Prototype-based Object Orientation:** JavaScript uses a prototype-based object model, where objects can inherit properties and methods from other objects. This allows for flexible object creation and behavior modification at runtime.

3. **First-class Functions:** In JavaScript, functions are treated as first-class citizens. They can be assigned to variables, passed as arguments to other functions, and returned as values from functions. This enables powerful functional programming techniques.

4. **Closures:** JavaScript supports closures, which allow functions to retain access to variables in their lexical scope, even after the outer function has finished executing. This enables encapsulation and the creation of private variables.

5. **Asynchronous Programming:** JavaScript provides mechanisms for asynchronous programming, such as callbacks, promises, and async/await. These allow for non-blocking execution of code, enabling efficient handling of tasks like network requests or file operations.

6. **Event-Driven Programming:** JavaScript has a strong focus on event-driven programming. It provides the ability to register event handlers and respond to user actions or other events triggered in the browser or the environment where JavaScript is running.

JavaScript Identifiers:

In JavaScript, identifiers are used to name variables, functions, objects, and other entities in the code.

Here are some rules and conventions regarding JavaScript identifiers:

1. **Naming Convention:** Identifiers in JavaScript are case-sensitive. It is common practice to use camelCase for naming variables and functions, where the first letter of each word is lowercase, and subsequent words are capitalized (e.g., `myVariable`, `calculateSum()`).

2. **Allowed Characters:** Identifiers can contain letters (both uppercase and lowercase), digits, underscores (_), or dollar signs ($). They must start with a letter, underscore, or dollar sign. However, it's recommended to use letters as the first character to ensure better compatibility and avoid potential issues.

3. **Reserved Keywords:** JavaScript has reserved keywords that cannot be used as identifiers because they have special meanings in the language. Examples of reserved keywords include `if`, `else`, `for`, `function`, and `var`. You cannot use these words as variable or function names.

4. **Best Practices:** It's important to choose meaningful and descriptive names for identifiers to enhance code readability and maintainability. Avoid using overly generic names like `x`, `y`, or `temp`. Instead, opt for descriptive names that convey the purpose or functionality of the entity.

Here's an example of using identifiers in JavaScript:

```javascript
// javascript
// Variable identifiers
var myVariable = 10;
var firstName = "John";

// Function identifier
function greet(name) {
  console.log("Hello, " + name + "!");
}
// Object identifier
```

```
var person = {
  name: "Alice",
  age: 25
};
// Using identifiers in code
greet(firstName);
console.log(person.name);
```

In the example above, `myVariable`, `firstName`, `greet`, and `person` are all identifiers used to name variables, functions, and objects in JavaScript.

Array with methods:

In JavaScript, arrays are a data structure used to store multiple values in a single variable. They come with several built-in methods that allow you to manipulate and perform operations on arrays. Here are some commonly used array methods:

```
// 1. ** push **: Adds one or more elements to the end of an array.
let fruits1 = ["apple", "banana"];
fruits1.push("orange");
// fruits is now ["apple", "banana", "orange"]
```

```
// 2. ** pop **: Removes the last element from an array and returns it.
let fruits2 = ["apple", "banana", "orange"];
let lastFruit = fruits2.pop();
// lastFruit is "orange"
// fruits is now ["apple", "banana"]
```

```
// 3. ** concat **: Concatenates two or more arrays and returns a new array.
let fruits3 = ["apple", "banana"];
let moreFruits = ["orange", "grape"];
let allFruits = fruits3.concat(moreFruits);
// allFruits is ["apple", "banana", "orange", "grape"]
```

```
// 4. ** join **: Joins all elements of an array into a string, with an
optional separator.
let fruits4 = ["apple", "banana", "orange"];
let fruitString = fruits4.join(", ");
```

```
// fruitString is "apple, banana, orange"
```

```
// 5. ** slice **: Returns a shallow copy of a portion of an array into a
new array.
let fruits5 = ["apple", "banana", "orange", "grape"];
let citrusFruits = fruits5.slice(2);
// citrusFruits is ["orange", "grape"]
```

```
// 6. ** indexOf **: Returns the index of the first occurrence of a
specified element in an array, or - 1 if not found.
let fruits6 = ["apple", "banana", "orange"];
let bananaIndex = fruits6.indexOf("banana");
// bananaIndex is 1
```

User-defined functions:

In JavaScript, you can define your own functions to encapsulate reusable blocks of code. User-defined functions allow you to perform specific tasks or calculations. Here's an example of a user-defined function:

```
function calculateSum(a, b) {
  return a + b;
}

let result = calculateSum(3, 5);
// result is 8
```

In the above example, `calculateSum` is a user-defined function that takes two parameters `a` and `b`. It calculates the sum of `a` and `b` and returns the result.

Predefined functions:

JavaScript also provides a range of built-in or predefined functions that perform common tasks or provide utility functionality. These functions are available without the need for explicit declaration. Some examples of predefined functions include:

```


// 1. ** parseInt **: Converts a string to an integer.
```

```
let number1 = parseInt("42");
// number is 42


// 2. ** parseFloat **: Converts a string to a floating - point number.
let number2 = parseFloat("3.14");
// number is 3.14

// 3. ** setTimeout **: Executes a function after a specified delay(in
milliseconds).
setTimeout(function() {
  console.log("Delayed function executed");
}, 2000); // Executes after 2 seconds

// 4. ** alert **: Displays an alert dialog box with a specified message.
alert("Hello, world!");

// 5. ** console.log **: Outputs a message or object to the web console.
console.log("Hello, world!");
```

These are just a few examples of the many predefined functions available in JavaScript. They provide various functionalities like mathematical calculations, string manipulations, date and time operations, DOM manipulation, and more.

```
```
In JavaScript, errors and exceptions are mechanisms used to handle and
manage unexpected or erroneous situations that can occur during the
execution of a program.When an error or exception occurs, it interrupts the
normal flow of the program and provides information about the issue
encountered.Here are some common types of errors and exceptions in
JavaScript:

1. ** Syntax Errors **: These occur when the code is not valid
JavaScript.This can happen when you have missing brackets, missing quotes,
or extra characters that are not valid in JavaScript.
2. ** Reference Errors **: These occur when the code references a variable
that is not defined.
3. ** Type Errors **: These occur when an operation is performed on a value
of an inappropriate type.
4. ** Range Errors **: These occur when a numeric value is not within the
acceptable range.
5. ** Custom Errors **: JavaScript also allows you to create custom errors
using the (Error) constructor.
```
```

```

```
1. ** Syntax Errors **: These occur when the code violates the syntax rules of the JavaScript language.It could be due to missing or misplaced characters, incorrect punctuation, or incorrect use of keywords.Syntax errors are usually detected by the JavaScript interpreter during the initial parsing of the code and prevent the code from running.

    Example:
```
let x = 5;
console.log(x;
// SyntaxError: missing ) after argument list

```
2. ** Reference Errors **: These occur when an invalid reference is made to a variable or function that does not exist or is out of scope.It typically happens when trying to access an undeclared variable or a variable that is not accessible in the current context.

    Example:
```
console.log(nonExistentVariable);
// ReferenceError: nonExistentVariable is not defined
```
3. ** Type Errors **: These occur when an operation is performed on a value of an inappropriate type.It can happen when trying to call a non - function, accessing properties on non - objects, or using incompatible operators.

    Example:
```
let num = 42;
num.toUpperCase();
// TypeError: num.toUpperCase is not a function

```
4. ** Range Errors **: These occur when a numeric value is not within the acceptable range.It usually happens when using methods like 'Array' or 'String' with invalid index values or specifying incorrect arguments for functions that expect a specific range of values.

    Example:
```
let arr = [1, 2, 3];
arr[10];
// RangeError: Invalid array length
```

```
5. ** Custom Errors **: JavaScript also allows you to create custom errors using the 'Error' constructor.Custom errors can be useful when you want to provide more specific information about an error or handle specific cases in your code.
    Example:
```
```
throw new Error("Custom error message");



```
To handle errors and exceptions, JavaScript provides the 'try...catch' statement, which allows you to attempt executing a block of code and catch any resulting errors or exceptions.This helps in gracefully handling errors and taking appropriate actions without abruptly terminating the program.

    Example:
```
```
try {
  // Code that might throw an error
  let result = someFunction();
} catch (error) {
  // Code to handle the error
  console.log("An error occurred:", error.message);
}

```
By using 'try...catch', you can capture and handle specific types of errors, perform error logging, display user - friendly error messages, or take alternative actions to recover from errors.
```

Control and Looping Structures

In JavaScript, control structures and looping structures allow you to control the flow of execution in your code and perform repetitive tasks. They help you make decisions, conditionally execute blocks of code, and iterate over data structures. Here are the main control and looping structures in JavaScript:

1. **if...else**: The `if...else` statement allows you to execute a block of code conditionally based on a specified condition.

```javascript
if (condition) {
  // Code to execute if the condition is true
} else {
  // Code to execute if the condition is false
}
```

Example:
```javascript
let num = 10;
if (num > 0) {
  console.log("The number is positive");
} else {
  console.log("The number is zero or negative");
}
```

2. **switch**: The `switch` statement allows you to execute different blocks of code based on different possible values of an expression.

```javascript
switch (expression) {
  case value1:
    // Code to execute if expression matches value1
    break;
  case value2:
    // Code to execute if expression matches value2
    break;
  default:
    // Code to execute if expression does not match any case
    break;
}
```

Example:

```javascript
let day = "Monday";
switch (day) {
  case "Monday":
    console.log("It's Monday");
    break;
  case "Tuesday":
    console.log("It's Tuesday");
    break;
  default:
    console.log("It's neither Monday nor Tuesday");
    break;
}
```

3. **for**: The `for` loop is used to iterate over a block of code for a specified number of times.

```javascript
for (initialization; condition; increment/decrement) {
  // Code to execute in each iteration
}
```

Example:
```javascript
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

4. **while**: The `while` loop is used to execute a block of code repeatedly as long as a specified condition remains true.

```javascript
while (condition) {
  // Code to execute in each iteration
}
```

Example:
```javascript
let i = 0;
while (i < 5) {
  console.log(i);
  i++;
}
```

5. **do...while**: The `do...while` loop is similar to the `while` loop, but it executes the block of code first and then checks the condition. It guarantees that the block of code will execute at least once.

```javascript
do {
  // Code to execute in each iteration
} while (condition);
```

Example:
```javascript
let i = 0;
do {
  console.log(i);
  i++;
} while (i < 5);
```

6. **for...in**: The `for...in` loop is used to iterate over the properties of an object.

```javascript
for (variable in object) {
  // Code to execute in each iteration
}
```

Example:
```javascript
let person = {
```

```
  name: "John",
  age: 30,
  profession: "Developer"
};

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

7. **for...of**: The `for...of` loop is used to iterate over iterable objects, such as arrays or strings.

```javascript
for (variable of iterable) {
  // Code to execute in each iteration
}
```

Example:
```javascript
let fruits = ["apple", "banana", "orange"];

for (let fruit of fruits) {
  console.log(fruit

);
}
```

These control and looping structures allow you to conditionally execute code, iterate over arrays or objects, and control the flow of execution based on certain conditions, providing flexibility and control in your JavaScript programs.