# JavaScript

CPIT-405 Web Applications

# Table of Contents

## Part 1: Fundamentals of JavaScript

## Introduction (I)

- JavaScript is the programming language of the web.
- ECMAScript (ES) is the specification that defines the core syntax, features, and functionalities of JavaScript.
- ECMAScript (ES) has versions while JavaScript does not.
  - JavScript is the implementation of the ECMAScript standard.
  - ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5, and ES6.
    - Since 2016, versions are named by year (ECMAScript 2016, 2017, 2018, 2019, 2020, etc.)
- JavaScript is an interpreted/JIT language
  - Traditionally, JavaScript engines in web browsers interpreted JavaScript code line by line, translated it into machine code on the fly.
  - However, modern JavaScript engines utilize Just-In-Time (JIT) compilation at run time to improve performance.

## Introduction (II)

- JavaScript was primarily a client-side scripting language executing on the user's web browser to create interactive and dynamic web pages.
- With the rise of server-side JavaScript environments (e.g., (Node.js ↗), JavaScript is also used for server-side scripting.
- Today, JavaScript is used in various development areas:
  - Web development: front-end and back-end (node.js).
  - Mobile app development: through frameworks like React Native ↗ and Ionic ↗.
  - Desktop app development: through frameworks like Electron.
  - Data science and machine learning: through libraries and frameworks like D3.js ↗, TensorFlow.js ↗, and Brain.js ↗.

## History (I)

- **1990**: Tim Berners-Lee created the first web browser called (WorldWideWeb and later renamed to Nexus).
- **1994**:
  - The Nexus browser was discontinued as more browsers evolved.
  - Netscape web browser was released and became the dominate web browser in the mid 1990s.
- **1995**:
  - JavScript was created by Brendan Eich at Netscape (now Mozilla) in the early days of the web.
  - Netscape added built-in support for JavaScript.
  - Microsoft released Internet Explorer.
- **1996**:
  - Netscape submitted the language to the European Computer Manufacturer’s Association (ECMA).
  - Microsoft created their own scripting language called JScript to compete with Netscape.
    - JScript was not fully compatible with ECMAScript and had additional features.

## History (II)

- **1997**:
  - ECMAScript 1 (ES1) released.
  - JavaScript became a trademark issued to Sun Microsystems (now Oracle).
- **1998-1999**: ES2 and ES3 released.
- **2000s**:
  - With the rise of other browsers, JScript usage continued to decline in favor of JavaScript.
  - Microsoft ended support for JScript in 2009.
  - ES5 was released in 2009.
  - ES6 was released in 2015.
    - ES6 added classes, modules, arrow functions, destructuring assignment, template literals, and more features making JavaScript a powerful general-purpose language.
- **Present**: ECMAScript continued to release annual updates to the specs of the JavaScript language.

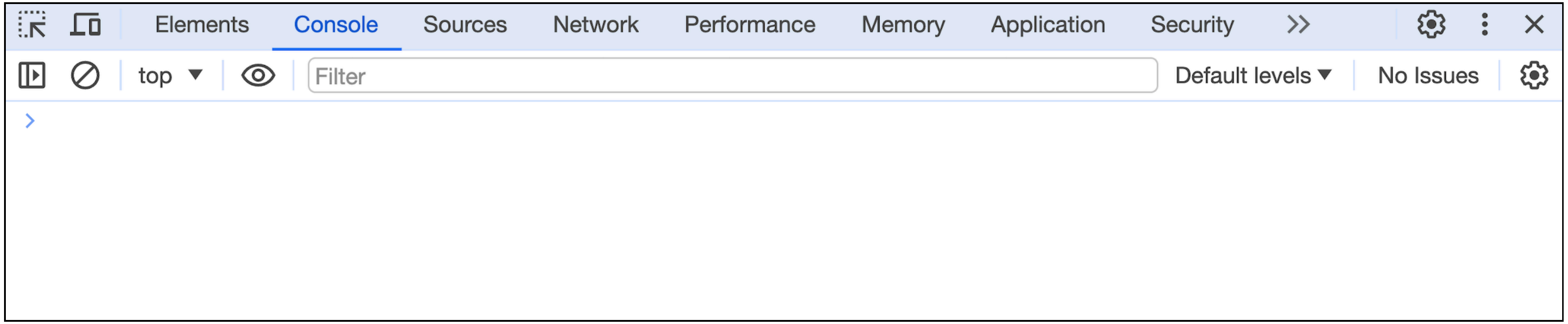
## Getting Started with JavaScript

You can write and execute JavaScript in multiple ways:

- Writing JavaScript in the Console
- Writing JavaScript in a web page
- Writing JavaScript in the terminal

## Writing JavaScript in the Console

- The easiest way to try JavaScript is to use the JavaScript console.
- The JavaScript console is a very useful tool to execute JavaScript code.
- All modern browsers have support for the console:
- Right-click anywhere on a webpage and select "Inspect" from the context menu. This will open the Developer Tools. Click on the Console tab.
- You can also open the console using the keyboard shortcut:
  - Windows: `F12` or `Ctrl`+`Shift`+`I`
  - macOS: `Command`+`Option`+`I`



## Writing JavaScript in a web page (I)

- In HTML, JavaScript code is inserted between `<script>` and `</script>` tags
- You can place the `<script>` tag in the `<head>` or `<body>` tags.
- You can also place JavaScript code in external files and reference it in your HTML document using `<script src="scriptFile.js"></script>`

```html
<!DOCTYPE html>
<html>
<head>
    <script>
        console.log("Hello JavaScript from within the head element!")
    </script>
</head>
<body>
    <h2>Hello JavaScript</h2>
    <script>
        console.log("Hello JavaScript from within the body element!")
    </script>
    <script src="externalJSFile.js"></script>
</body>
</html>
```

## Writing JavaScript in a web page (II)

- If placed inside the `<head>` tag, JavaScript code will be executed prior to the page being rendered by the browser. This can impact performance of loading and rendering your page.
- An external JavaScript file can be imported using the `src` attribute of the `<script>` tag.

  ```
  <script src="./path/to/script.js"></script>
  ```

  - Using external JavaScript files provides a better separation of concerns between HTML and code.
  - External JavaScript files can be cached by the browser to speed up page rendering time.

## Writing JavaScript in the Terminal

- You can also write and run JavaScript code from the command line:
- Install Node.js from nodejs.org ↗.
- Open your Terminal app or PowerShell on Windows
  - You can also save a file, navigate to its directory using `cd`, and execute the JavaScript file using

```
node script.js
```

  - Or You can run the interactive shell using the `node` command:

```
node
```

```
Welcome to Node.js v20.3.1.
Type ".help" for more information.
> console.log("hello JavaScript")
hello JavaScript
```

## Syntax

- **Case sensitivity**: JavaScript is a case-sensitive language.
  - Identifier with uppercase and lowercase letters are treated as distinct
- **Statements**: JavaScript programs are composed of a sequence of statements. A statement might declare a variable, loop over items, call a function, or change the value of a variable
- **Semicolons**: In JavaScript, semicolons are usually placed at the end of a statement.
  - However, JavaScript does not enforce the use of semicolons as it has automatic semicolon insertion (ASI) mechanism. If you omit a semicolon, the JavaScript engine will insert it for yous
- **Variables**: Variables are declared using `var`, `let`, or `const`. More on this later.
- **Data Types**: JavaScript has the following data types: Number, BigInt, String, Symbol, Boolean, Object, Null, and Undefined.
- **Comments**: JavaScript supports two styles of comments:
  - `// this is a single-line comment`
  - `/* This is a multi-line comment */`

## `var` variables

```
var x = 10;
if (x == 10) {
  var x = 20;
  console.log(x);
}
console.log(x);
```

```
20
20
```

## `let` variables

```
let x = 10;
if (x == 10) {
  let x = 20;
  console.log(x);
}
console.log(x);
```

```
20
10
```

## `var` Variables

- Using `var` for function-scoped or globally-scoped variables. This means:
  - a variable declared with var is accessible within the function it's declared in.
  - a variable declared with var outside any function is considered a global variable and is added to the global object `window`.

## `let` Variables

- Using `let` for re-assignable, block-scoped local variables. This means:
  - a variable declared with `let` is only accessible within the block it's declared in (e.g., within a for loop or an if statement).
  - a variable declared with `let` outside any function is considered a global variable in the sense that it can be accessed from anywhere in the script but won't be added to the global object `window`.

## Constants

- Using `const` for variables whose values must not be changed.

```
const discount = 0.20;
discount = 0.50; // Uncaught TypeError: Assignment to constant variable.
console.log(discount); // prints 0.20
```

```
0.20
```

- If a constant is an object, its properties can be updated, added, or removed. More on this later when discussing objects.

```
const course = {name: "CPIT-405", semester: "fall", year: 2023}
course.semester = "spring"
console.log(course) // prints
```

```
{ name: 'CPIT-405', semester: 'spring', year: 2023 }
```

## Primitive Data Types

| Type | Meaning | Example |
|---|---|---|
| Null | An absent of any object value | `let foo = null;` |
| Undefined | An absent of a value (variable declared but not assigned) | `let foo;` |
| Boolean | A boolean value `true` or `false` | `let foo = true;` |
| Number | A floating-point number | `let foo= 37; let bar=-9.14;` |
| Bigint | Too large numbers. | `let foo= = BigInt(Number.MAX_SAFE_INTEGER + 2);` |
| String | Represents textual data | `let foo = "A string primitive";` |
| Symbol | A unique and immutable value | `let foo= Symbol("foo");` |

## Primitive Data Types Example

- The `typeof` operator returns a string indicating the type of the argument.

```javascript
let age = 30; // Number (integer)
console.log(age, typeof age)

let pi = 3.14159; // Number (floating-point)
console.log(pi, typeof pi)

let name = "Ali Ahmed"; // String
console.log(name, typeof name)

let isStudent = true; // Boolean
console.log(isStudent, typeof isStudent)

let nothing = null; // Null
console.log(nothing, typeof nothing)

// Undefined (variable declared but not assigned)
let notAssigned;
console.log(notAssigned, typeof notAssigned)
```

```javascript
let uniqueSymbol = Symbol("unique"); // Symbol (unique and immutable identifier)
console.log(uniqueSymbol, typeof uniqueSymbol)

let bigInt = 9007199254740991n; // BigInt (large integer)
console.log(bigInt, typeof bigInt)
```

## Non-primitive Data Type (Object)

- **Object** is the only non-primitive data type in JavaScript.
- **Key-Value Pairs**: An object is a collection of key-value pairs
  - Keys are unique identifiers (usually strings or symbols)
  - Values can be any data type, including other objects, arrays, functions, or primitive types.
- **Unordered**: The order of key-value pairs within an object is not important and there is no guarantee that it will be preserved.
- **Mutable**: Objects in JavaScript are mutable. This means that once an object is created, its properties and values can be modified.
- **Methods**: Objects can contain functions called methods, which are used to perform operations on the object’s data.
- **Prototypal inheritance**: Objects may inherit properties and methods through a mechanism called prototypal inheritance. This allows object defined in another object called the prototype object.

## Object Example (I)

```
let person = {
  firstName: "Ali",
  lastName: "Ahmed",
  age: 21,
  isEnrolled: true,
  greet: function() {
    console.log("Hello, my name is " + this.firstName + " " + this.lastName);
  },
};
console.log(typeof person) // object
console.log("First Name: ", person.firstName, typeof person.firstName);
console.log("Age: ", person.age, typeof person.age)
console.log("Is enrolled:", isEnrolled, typeof person.isEnrolled);
person.greet();
console.log(typeof person.greet)
```

```
let course = {
  title: "Web Applications",
  instructor: {
    firstName: "Khalid",
    lastName: "Alharbi",
    year: 2024,
    address: null,
    getFullName: function() {
      return this.firstName + " " + this.lastName;
    }
  },
  duration: 15,
  isActive: true,
  tags: ["javascript", "react", "web development"],
  students: [{
      name: "Ali",
      gpa: 4.6,
      coursesTaken: [405, 498, 490]
    },
    {
      name: "Ahmed",
      gpa: 4.4,
      grades: [305, 405, 260]
    }
```

- Get the course title?

  `course.title`

- Get the instructor full name?

  `course.instructor.getFullName()`

- Get the name of the second enrolled student?

  `course.students[1].name`

- Enroll a new student in the course

  `course.enroll("Abdullah")`

## `null` ~~is an Object~~ 🙁 😱

- You may have noted in the previous example that after running

```
let nothing = null;
console.log(nothing, typeof nothing)
```

The output was:

```
null object
```

- It's important to clarify that **null is not actually an object** in JavaScript!
- It's a primitive data type that represents the intentional absence of any object value.
- The statement `typeof null` returns "object" due to a historical design choice in JavaScript that remains inaccurate.
  - It was initially intended to have `typeof null` return a different value for null, but the decision wasn't implemented.
  - It hasn't been changed due to backward compatibility concerns.

## Strings

- Strings can be created using either single quotes (`'`), double quotes (`"`), or backticks (`` ` ``).
- Backticks are used to define template literals, which allow embedded expressions and multi-line strings.
  - String interpolation can be done using template literals `Text and ${variable_name}`
- The `.length` property is used to find the length of a string.
- String methods include `charAt()`, `concat()`, `includes()`, `indexOf()`, `slice()`, `split()`, `substring()`, and `toUpperCase()`, `substr()`, `replace`, among others.
- Strings are immutable in JavaScript, which means they cannot be changed once they are created. However, when you "change" a string, you're actually creating a new string with the changes. The original string remains unmodified.
  - `let name = "Sami"; name[0]="R"; // name is unchanged (Sami)`
- JavaScript uses Unicode character set, which means strings can include characters from virtually any language and a variety of symbols.
- JavaScript provides escape sequences for special characters, such as `\'`, `\"`, `\\`, `\n`, `\t`, etc.

## Strings Example

```javascript
let name = 'Khalid';
console.log(`Hello, ${name}`); // Outputs: Hello, Khalid

// Creating strings
let string1 = 'Hello,';
let string2 = " world!";
let string3 = ` This is a string in JavaScript.`;

// Concatenating strings
let greeting = string1 + string2 + string3;
console.log(greeting);  // Outputs: Hello, world! This is a string in JavaScript.

// String length with string interpolation (template literals)
console.log(`Length of greeting: ${greeting.length}`);  // Outputs: Length of greeting: 45

// String methods
console.log(greeting.toUpperCase());  // Outputs: HELLO, WORLD! THIS IS A STRING IN JAVASCRIPT.
console.log(greeting.includes('JavaScript'));  // Outputs: true
console.log(greeting.indexOf('world'));  // Outputs: 7
console.log(greeting.slice(7, 12));  // Outputs: world

// Strings are immutable
greeting[0] = 'K';
console.log(greeting);  // Outputs: Hello, world! This is a string in JavaScript. (no change)
```

## Number

- The *Number* data type represents a numeric value, including integers (whole numbers) and floating-point numbers (decimals).
- It also includes special values like `Infinity`, `-Infinity`, and `NaN` (Not a Number).
- `NaN` is a special value representing an error or undefined result in numeric operations.
- Number uses a 64-bit double-precision floating-point format.
- `BigInt` data type can be used for values which are too large to be represented by the number primitive.
  - Useful in computation that involves extremely large integers for applications in finance and cryptography.

```
// Dividing positive number by zero returns Infinity
console.log(10/0)
// Dividing negative number by zero returns -Infinity
console.log(-3/0)
// Dividing zero by zero returns the type NaN
console.log(0/0);
// Converting a non-numeric string to a number using the Number constructor returns NaN
console.log(Number("zero"))
```

## Operators

- Arithmetic operators: `+`, `-`, `*`, `/`, `%` (modulo - remainder after division), `**` (exponentiation).
- Assignment Operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=`
- Bitwise operators: `>>`, `<<`, `&`, `|`, `^`, `~`
- Unary negation (`-`)
- Increment/decrement: `++`, `--`
- String operators
  - `+` (concatenation) `-` Joins two strings together.
- Comparison Operators: `==`, `!=`, `===`, `!==`, `<`, `>`, `<=`, `>=`
  - `==` and `!=` loose equality (compare values): will perform a type conversion when comparing two values
  - `===` and `!==` strict equality (compare values and types): will perform comparison but without type conversion.
  - The spread operator `...` to unpack array elements into individual elements

## `==` VS `===`

- Loose equality `==` vs strict equality `===`

```
if (5 == "5") // evaluates to true
   console.log("JavaScript will convert the operands to a common type through type coercion and perform comparison.")

if (5 === "5") // evaluates to false
   console.log("No")
else {
    console.log("JavaScript does not perform type coercion")
}
```

## Math Object

- The `Math` object is a built-in object in JavaScript that has properties and methods for mathematical constants and functions.

| Method | Description | Example |
|---|---|---|
| `Math.round()` | Rounds a number to the nearest integer | `Math.round(4.7);` |
| `Math.sqrt()` | Returns the square root of a number | `Math.sqrt(16);` |
| `Math.abs()` | Returns the absolute value of a number | `Math.abs(-5);` |
| `Math.random()` | Returns a random number between 0 (inclusive) and 1 (exclusive) | `Math.random();` |
| `Math.max()` | Returns the largest of zero or more numbers | `Math.max(5, 10);` |

## Date Object

- JavaScript provides the `Date` object for working with dates and times.
- You can create a new `Date` object with `new Date()`, which returns the current date and time.
- The `Date` object automatically uses the environment's time zone and can handle time zone conversion.
- The `Date` object provides methods for getting and setting each component of the date and time (year, month, day, hour, minute, second, millisecond).
  - Example methods: `getFullYear()`, `getMonth()`, `getDate()`, `getHours()`, `getMinutes()`, `getSeconds()`, and `getMilliseconds()`.
- The `Date` object provides methods to format dates as strings.
  - Example methods: `toDateString()`, `toTimeString()`, `toLocaleDateString()`, and `toLocaleTimeString()`
- JavaScript counts months from 0 to 11. January is 0, and December is 11.
- You can compare two dates using standard comparison operators. The dates are converted to milliseconds and then compared.

## Date and Time Example

### Date and Time Example

```javascript
// Creating a new Date object for the current date and time
let now = new Date();
console.log(now);

// Creating a Date object for a specific date
let specificDate = new Date('2024-02-03T00:00:00');
console.log(specificDate);

// Getting components of the date
console.log(`Year: ${now.getFullYear()}`);
console.log(`Month: ${now.getMonth()}`);

// Formatting the date as a string
console.log(`Date string: ${now.toDateString()}`);
console.log(`Time string: ${now.toTimeString()}`);
console.log(`Locale date string: ${now.toLocaleDateString()}`);
console.log(`Locale time string: ${now.toLocaleTimeString()}`);
```

### Comparing Dates Example:

```javascript
// Comparing the dates
if (specificDate < now) {
    console.log('specificDate is less than now');
} else if (specificDate > now) {
    console.log('specificDate is greater than now');
} else {
    console.log('specificDate is equal to now');
}
```

## Arrays (I)

- An array in JS is an object that represents a list of elements. Commons array methods include:

| Function | Description | Example |
|---|---|---|
| `push()` | Adds one or more elements to the end of an array and returns the new length of the array. | `let arr = [1, 2]; arr.push(3);` |
| `pop()` | Removes the last element from an array and returns that element. | `let arr = [1, 2, 3]; arr.pop();` |
| `join()` | Joins all elements of an array into a string. | `let arr = ['Hello', 'World']; let str = arr.join(' ');` |
| `concat()` | Returns a new array that is this array joined with other array(s) and/or value(s). | `let arr1 = [1, 2]; let arr2 = [3, 4]; let newArr = arr1.concat(arr2);` |

## Arrays (II)

| Function | Description | Example |
|---|---|---|
| `forEach()` | Calls a function for each element in the array. | `let arr = [1, 2, 3];` `arr.forEach(x => console.log(x));` |
| `sort()` | Sorts the elements of an array in place and returns the array. | `let arr = [3, 1, 2]; arr.sort();` |
| `reverse()` | Reverses the order of the elements of an array in place. | `let arr = [1, 2, 3];` `arr.reverse();` |
| `indexOf()` | Returns the first (least) index of an element within the array equal to the specified value, or -1 if none is found. | `let arr = [1, 2, 3]; let index =` `arr.indexOf(2);` |

## Arrays (III)

```javascript
// Creating an array
let fruits = ['Apple', 'Banana', 'Cherry'];
console.log(fruits);

// Accessing elements of an array
console.log(fruits[0]);
console.log(fruits[1]);
console.log(fruits[2]);

// Getting the length of an array
console.log(fruits.length);

// Adding elements to an array
fruits.push('Date');
console.log(fruits);

// Removing the last element from an array
let lastFruit = fruits.pop();
console.log(lastFruit);
console.log(fruits);
```

```javascript
// get a new string with all elements are
// joined by the given separator
fruits.join(', ') // Apple, Banana, Cherry

// Reverse the order of the elements
let reversed = fruits.reverse();
console.log(reversed); // [ 'Cherry', 'Banana', 'Apple' ]

// Finding the index of an element in an array
let index = fruits.indexOf('Cherry');
console.log(index);

// Combining the existing fruits array with a new one
fruits = fruits.concat(['Grape', 'Pomegranate'])
console.log(fruits);

// Looping over an array
fruits.forEach(function(fruit, index) {
    console.log(`Fruit at index ${index} is ${fruit}`);
});
```

## The Spread Operator `...` (I)

- The spread operator is `...` is used to spread or unpack elements of an iterable (array or object) into individual elements.
- It allows expressions to be expanded in places where multiple arguments or variables are expected.
- For example, instead of writing this:

```
let smallArr = [1,2,3]
let bigArr = [smallArr[0], smallArr[1], smallArr[2], 4, 5, 6]
console.log(bigArr);
```

```
1, 2, 3, 4, 5, 6
```

- We can use `...smallArr` to spread the elements of *smallArr* into the big array, *bigArr*.

```
let smallArr = [1,2,3]
let bigArr = [...smallArr, 4, 5, 6]
```

```
1, 2, 3, 4, 5, 6
```

## Using the spread operator `...` in a function call

```
let fruits = ["apple", "orange", "banana"]
console.log(...fruits)
```

```
apple orange banana
```

## Using the spread operator `...` to pass array elements as arguments to a function

```
function add(x, y, z){
  return x + y + z;
}
let numbers = [2, 4, 6]
let total = add(...numbers)
console.log(total)
```

```
12
```

## Using the spread operator `...` with Objects

```javascript
let orange1 = {
  type: "Orange",
  skinColor: "Orange",
  sweet: false
};

let orange2 = {
  citrus: true,
  vitaminC: "High"
};

let orange = {...orange1, ...orange2, edible: true, canBeJuiced: true}

console.log(orange)
```

```
{
  type: 'Orange',
  skinColor: 'Orange',
  sweet: false,
  citrus: true,
  vitaminC: 'High',
  edible: true,
  canBeJuiced: true
}
```

## Conditional statements

- **if...else** : Executes different code blocks based on a condition.

```
if (condition1)
  statement1
else if (condition2)
  statement2
else
  statementN
```

- **switch**: Executes different code blocks based on the value of an expression compared to multiple cases.

```
switch (expression) {
  case caseExpression:
    statements
  default:
    statements
}
```

- **Conditional (ternary) operator `?` (also known as inline if statement)**: shorthand for an `if...else` statement

```
condition ? exprIfTrue : exprIfFalse
```

## if-else statement example

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

```
The number is positive.
```

## switch statement example

```
let day = 2;
switch (day) {
  case 1:
    console.log("Today is Sunday.");
    break;
  case 2:
    console.log("Today is Monday.");
    break;
  default:
    console.log("Unsupported day number.");
}
```

```
Today is Monday.
```

## Conditional (ternary) operator example

```
// Conditional (ternary) operator ?
let isMember = false;
let fee = isMember ? '$2.00' : '$10.00';
console.log(fee);
```

```
$10.00
```

## Iteration statements

- `for`: Repeats a block of code a specific number of times.
- `forEach`: is not a statement but a method to iterate over the elements of an array.
- `while`: Repeats a block of code as long as a condition is true.
- `do...while`: Executes a block of code at least once, then checks a condition to repeat.
- `for...of`: Iterates over the values of an iterable object (e.g., arrays, strings).
- `for...in`: Iterates over the index of an array or the properties of an iterable object (e.g., arrays, strings).

## for loop example

```
let array = [1, 2, 3, 4, 5];

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

```
1
2
3
4
5
```

## `forEach` method example

```
let array = [1, 2, 3, 4, 5];

array.forEach(function(element) {
  console.log(element);
});
```

```
1 2 3 4 5
```

## `while` and `do..while`

```
let count = 1;

while (count <= 5) {
  console.log(count++);
}

count = 1;

do {
  console.log(count++);
} while (count<=5);
```

```
1
2
3
4
5


1
2
3
4
5
```

## `for..of` and `for..in`

```
let fruits = ["Apple", "Orange", "Banana"]

for (let f of fruits) {
  console.log(f);
}

for (let i in fruits) {
  console.log(fruits[i]);
}
```

```
Apple
Orange
Banana

Apple
Orange
Banana
```

## Functions

- A function is a reusable and self-contained block of code.
- The code inside a function is not executed when the function is declared but rather when it is invoked.

```
// function declaration
function add(x, y){
  return x+y;
}
// function invocation
console.log(add(3,2));
```

- In JavaScript, functions are considered first class citizens:
  - Functions are not required to be declared in a class.
  - Functions can be assigned to a variable and treated like values of other types
  - Functions can be passed to other functions as parameters
  - Functions can return a primitive-data type, objects, or even another function
  - Functions are essentially objects but with an additional capability of being invoked/executed.

## Functions declaration

There are multiple ways to declare functions in JavaScript:

1. Function Declaration (or Function Statement)
2. Function Expression (or assigning an anonymous function to a variable)
3. Named Function Expression
4. Arrow Function
5. Immediately Invoked Function Expression (IIFE)

## 1. Function Expression (or Function Statement)

```
function getCourse() {
  console.log("CPIT 405");
}
// invoke the function
getCourse();
```

```
CPIT-405
```

## 2. Function Expression (or assigning an anonymous function to a variable)

```
let getCourse = function() {
  console.log("CPIT-405);
};

// invoke the function
getCourse();
```

```
CPIT-405
```

## 3. Named Function Expression

```
let getMyCourse = function getCourse() {
  console.log("CPIT 405");
}
// invoke the named function
getMyCourse();
// invoking getCourse will result in an error
getCourse();
```

```
CPIT-405
Uncaught ReferenceError: getCourse is not defined
```

## 4. Arrow Function

```
let getCourse = () => {
  console.log("CPIT 405");
}

let getDepartment = () => return "IT";

let getCollege = () => "FCIT";

let getUniversity = () => {
  let university = "KAU";
  return university
}
// invoke the above arrow functions
getCourse();
getDepartment();
getCollege()
getUniversity();
```

```
CPIT-405
IT
FCIT
KAU
```

## 5. Immediately Invoked Function Expression (IIFE)

- An IIFE (Immediately Invoked Function Expression) is a JavaScript function that runs as soon as it is defined.

```
(function() {
  console.log("CPIT-405");
})();
```

```
CPIT-405
```

## Function invocation

- Function invocation means calling or executing a function.
- In JavaScript, functions are often invoked by appending parentheses `()` to the function name.

```
function getUniversity(){
  return "KAU";
}
console.log(getUniversity());
```

```
let person = {
  firstName: "Muhammad",
  lastName: "Ahmed",
  fullName: function() {
    return this.firstName + " " + this.lastName
  }
}
console.log(person.fullName())
```

```
KAU
Muhammad Ahmed
```

## Nested Functions

- In JavaScript, functions may be nested within other functions.

```
function addSquares(a, b) {
  function square(x) {
    return x * x;
  }
  return square(a) + square(b);
}

console.log(addSquares(2, 3));
```

```
13
```

## Handling Extra Arguments

- Passing additional arguments to a function doesn't cause an error.
- However, these extra arguments are not utilized by the function.
- Consider the following example:

```
function sum(x, y, z){
  return x + y + z;
}

let total = sum(1, 2, 3, 4, 5, 6)
console.log(total)
```

```
6
```

- In this case, the sum function only uses the first three arguments (1, 2, 3). The rest of the arguments (4, 5, 6) are ignored.
- This is where the rest parameter operator (...) comes in handy. It allows a function to accept any number of arguments and use them all.

## The rest parameters `...`

- The rest parameters syntax `...` allows a function to accept an indefinite number of arguments as an array
- They are used in function definitions to bundle arguments into an array
- The rest parameters must be the last argument in a function’s parameter list.
- Rest parameters can be used as optional arguments in a function.

```
function sum(n, m, ...numbers) {
    let sum = n + m;
    for (let i=0; i<numbers.length; i++){
      sum += numbers[i]
    }
    return sum
}

let total = sum(1,2,3,4); // n=1, m=2, numbers=[3,4]
console.log(total);
total = sum(1,2)
```

```
10
3
```

## `...` Syntax: Rest Parameters vs Spread Operator

- Both the spread operator and rest parameters use the same syntax (`...`), but they are used in different contexts and serve different purposes.
- When ... precedes an array or an object variable, it acts as the spread operator.
  - The spread operator is used to unpack iterable objects (strings, arrays, and objects) into individual elements.
  - Example: `let arr1 = [1, 2, 3]; let arr2 = [...arr1];`
- When ... is used in the function declaration as a parameter variable, it signifies the rest parameters.
  - The rest parameters are used in function definitions to represent an indefinite number of arguments as an array.
  - Example: `function sum(...numbers) {console.log(numbers.length);}`

## Part 2: Mouse and Keyboard Events

## Mouse Events

- Mouse events are triggered when using a pointer device like the mouse. The most common events are: `click`, `dbclick`, `mouseup` and `mousedown`.

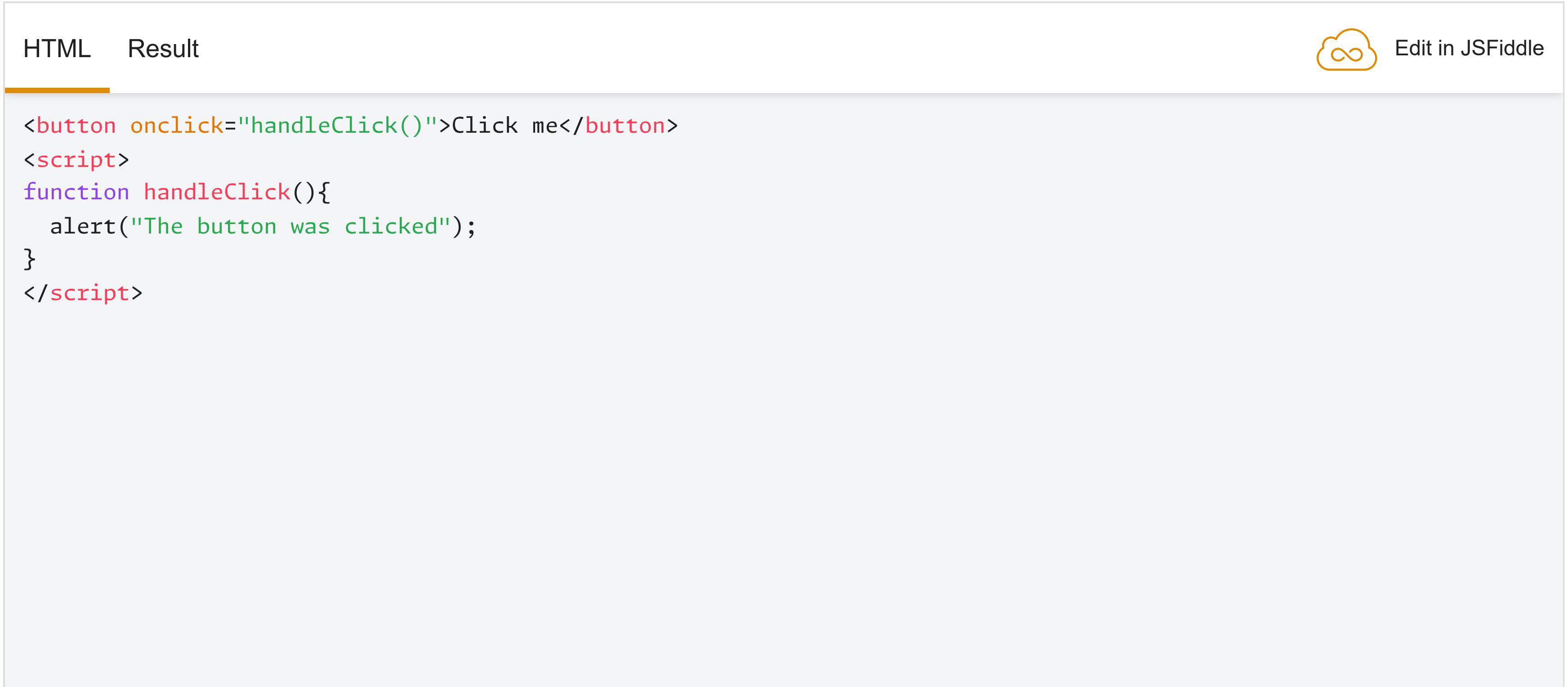
There are two approaches for registering mouse or any event handlers in general:

1. inline in the HTML using the **onevent property** (e.g. `onclick` for the click event).

2. Dynamically in JS by adding the event handler to the element in the DOM tree using the `addEventListener()` method.

## 1) Registering Mouse Events Inline (in HTML)

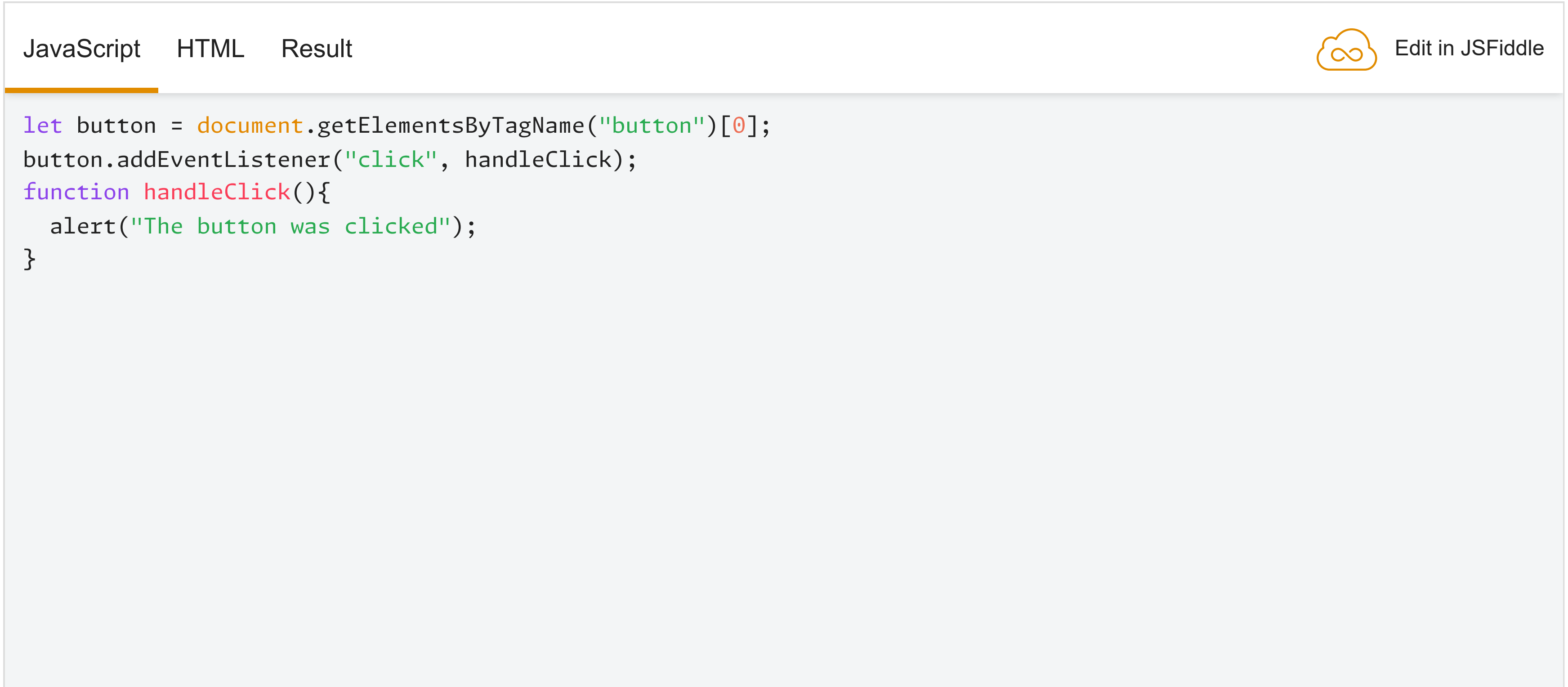
- We can add mouse events inline in HTML using the *onevent* property. For example, the click event can be registered using the `onclick` property and assigning it to the a function **with parentheses for invocation** as shown in the following example:



```html
<button onclick="handleClick()">Click me</button>
<script>
function handleClick(){
  alert("The button was clicked");
}
</script>
```

## 2) Registering Mouse Events in the DOM (in JS)

- We can register mouse events in *JavaScript* using `addEventListener` of the DOM element and assigning it to the function name **without parentheses** so it will be invoked when the event is fired inside the *addEventListener* function as in the following example:



```javascript
let button = document.getElementsByTagName("button")[0];
button.addEventListener("click", handleClick);
function handleClick(){
  alert("The button was clicked");
}
```

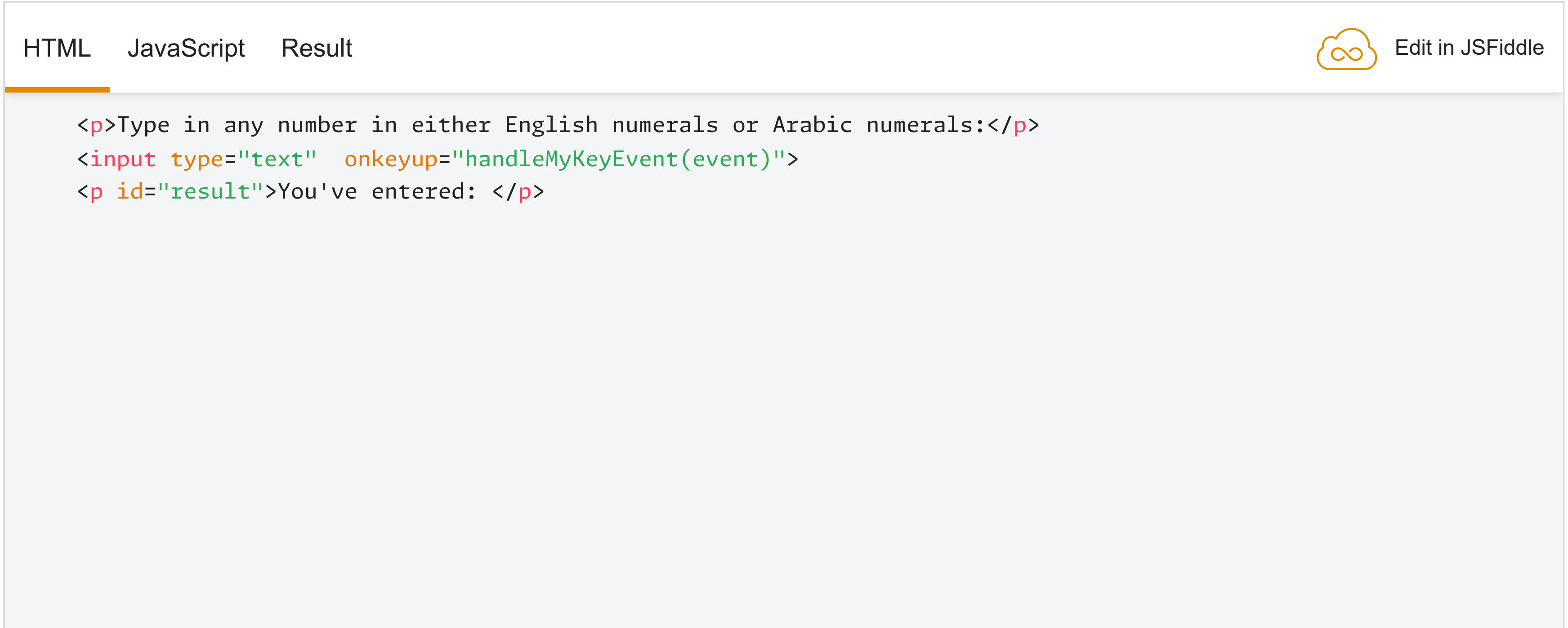
## Keyboard events

- Keyboard events are triggered when interacting with the keyboard. There are two keyboard events: `keydown` (a key has been pressed) and `keyup` (a key has been released). Similar to mouse events, there are two approaches for registering keyboard or any event handlers in general:

1. Inline in the HTML using the **onevent property** (e.g. `onkeydown` for pressing a key and `onkeyup` for releasing a key).
2. Dynamically in JS by adding the event handler to the element in the DOM tree using the `addEventListener()` method.

## 1) Registering Keyboard Events Inline (in HTML)

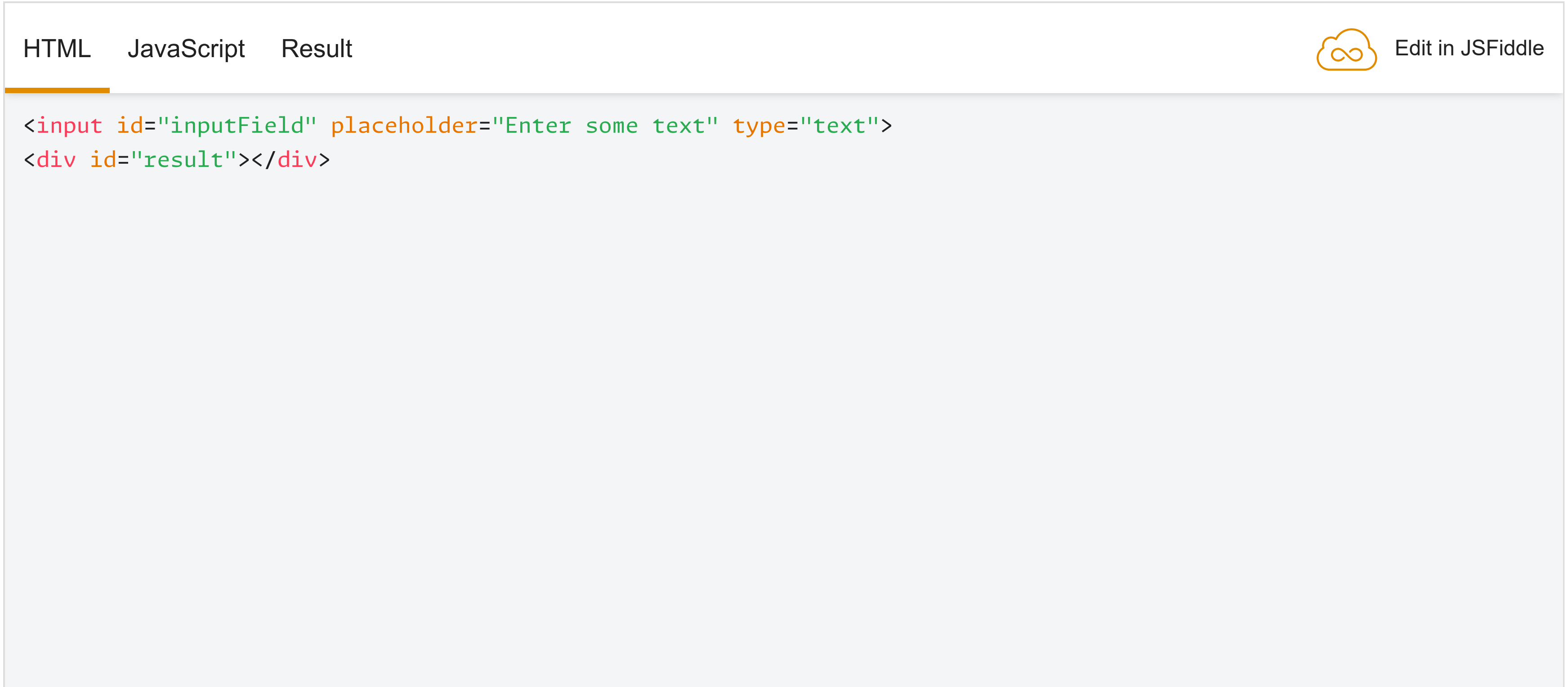
- We can add keyboard events inline in HTML using the *onevent* property. For example, the *key up* event can be registered using the `onkeyup` property and assigning it to the a function **with parentheses for invocation.** The following example uses the `onkeyup` event to listen to key press and release events to convert numbers entered in Arabic numerals to English numerals.



```html
<p>Type in any number in either English numerals or Arabic numerals:</p>
<input type="text"  onkeyup="handleMyKeyEvent(event)">
<p id="result">You've entered: </p>
```

## 2) Registering Keyboard Events in the DOM (in JS)

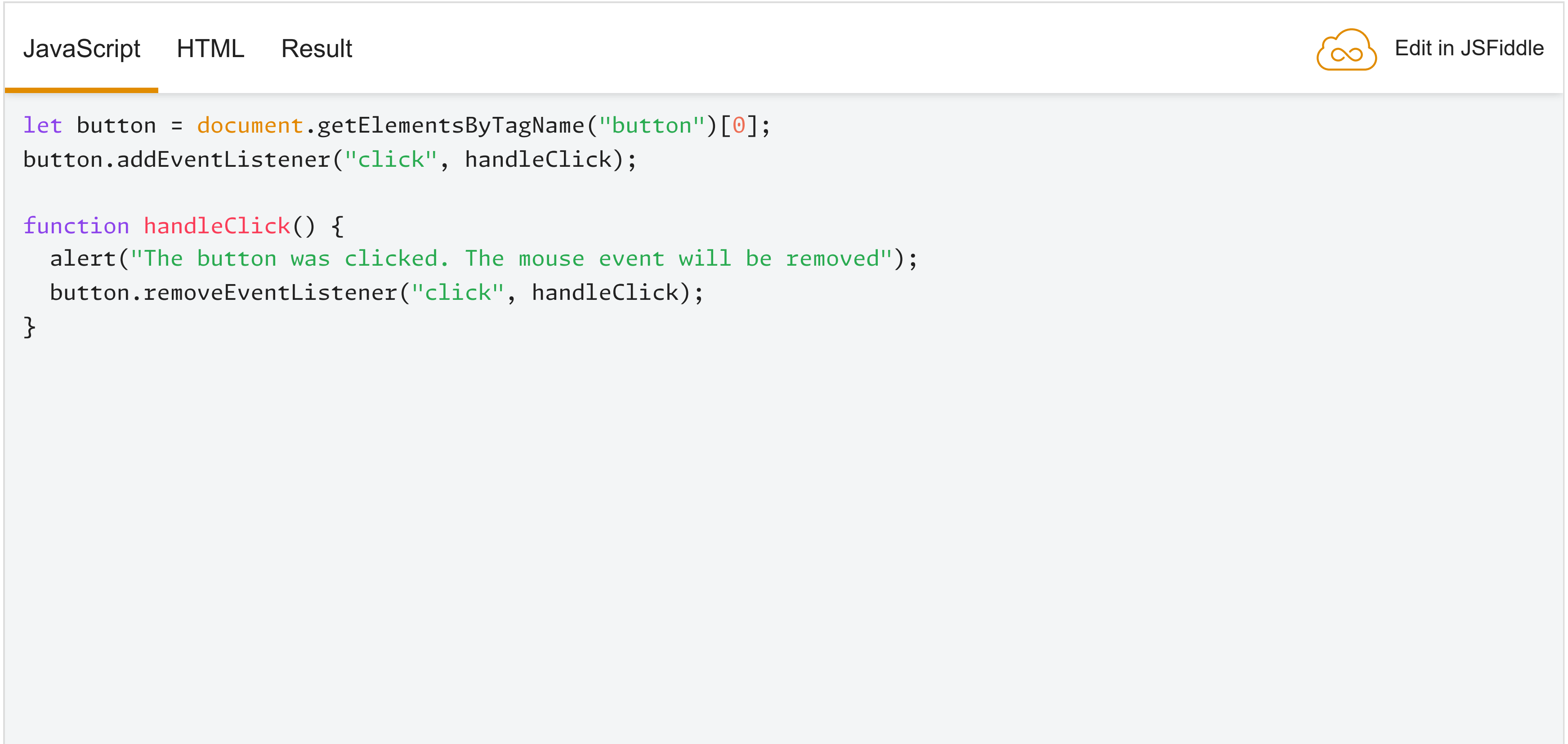
- Similar to registering mouse events, we can register keyboard events in *JavaScript* using `addEventListener` of the DOM element and assigning it to the function name **without parentheses** so it will be invoked when the event is fired inside the *addEventListener* function as in the following example:



```html
<input id="inputField" placeholder="Enter some text" type="text">
<div id="result"></div>
```

## Removing Registered Mouse or Keyboard Events

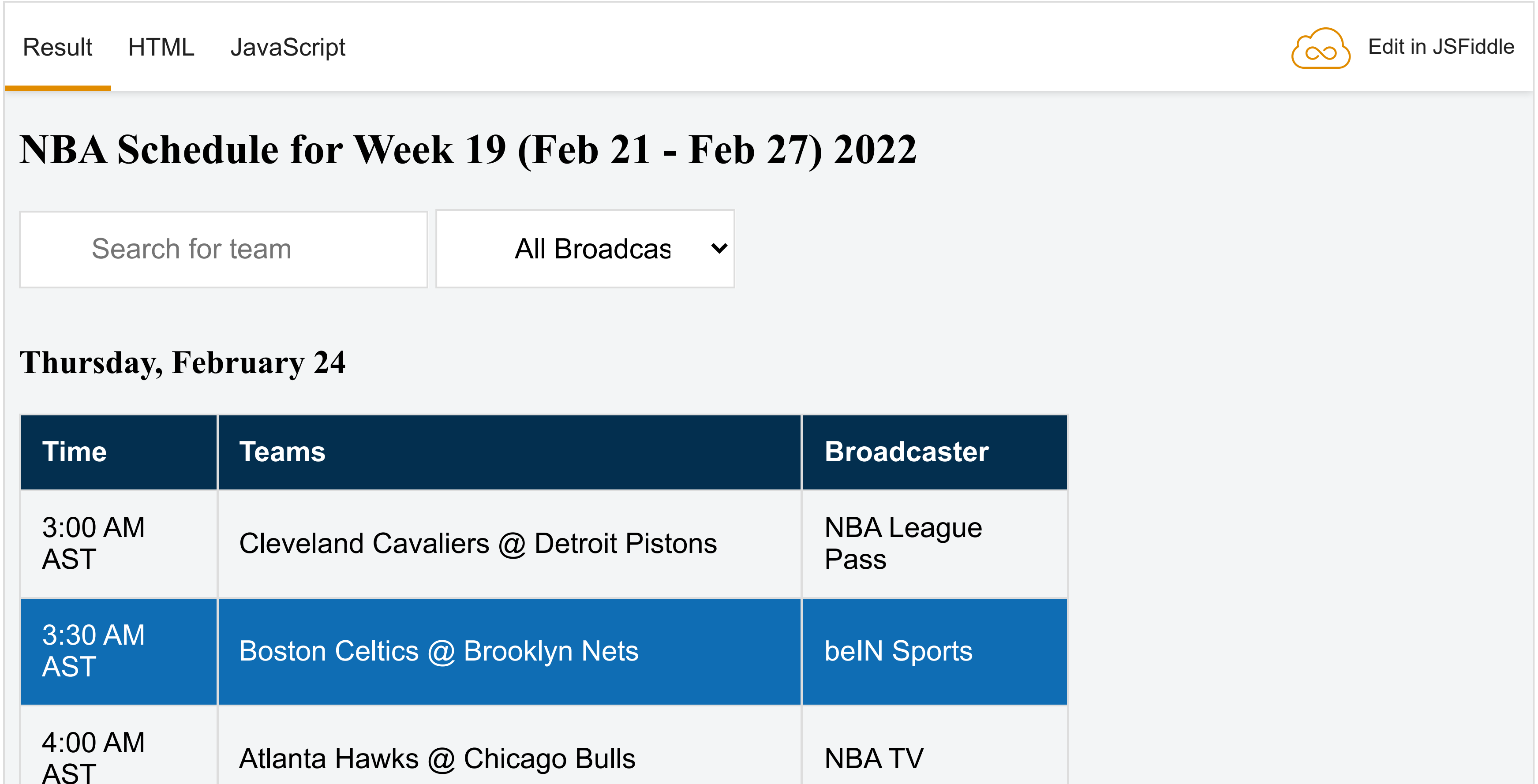
- We can remove registered mouse or keyboard events using `removeEventListener` as shown in the following example:



```javascript
let button = document.getElementsByTagName("button")[0];
button.addEventListener("click", handleClick);

function handleClick() {
  alert("The button was clicked. The mouse event will be removed");
  button.removeEventListener("click", handleClick);
}
```

## Example 1: Complete Mouse and Keyboard Events

- The following example shows how to filter a table by a word and/or by an item in a drop down menu.



**NBA Schedule for Week 19 (Feb 21 - Feb 27) 2022**

Search for team

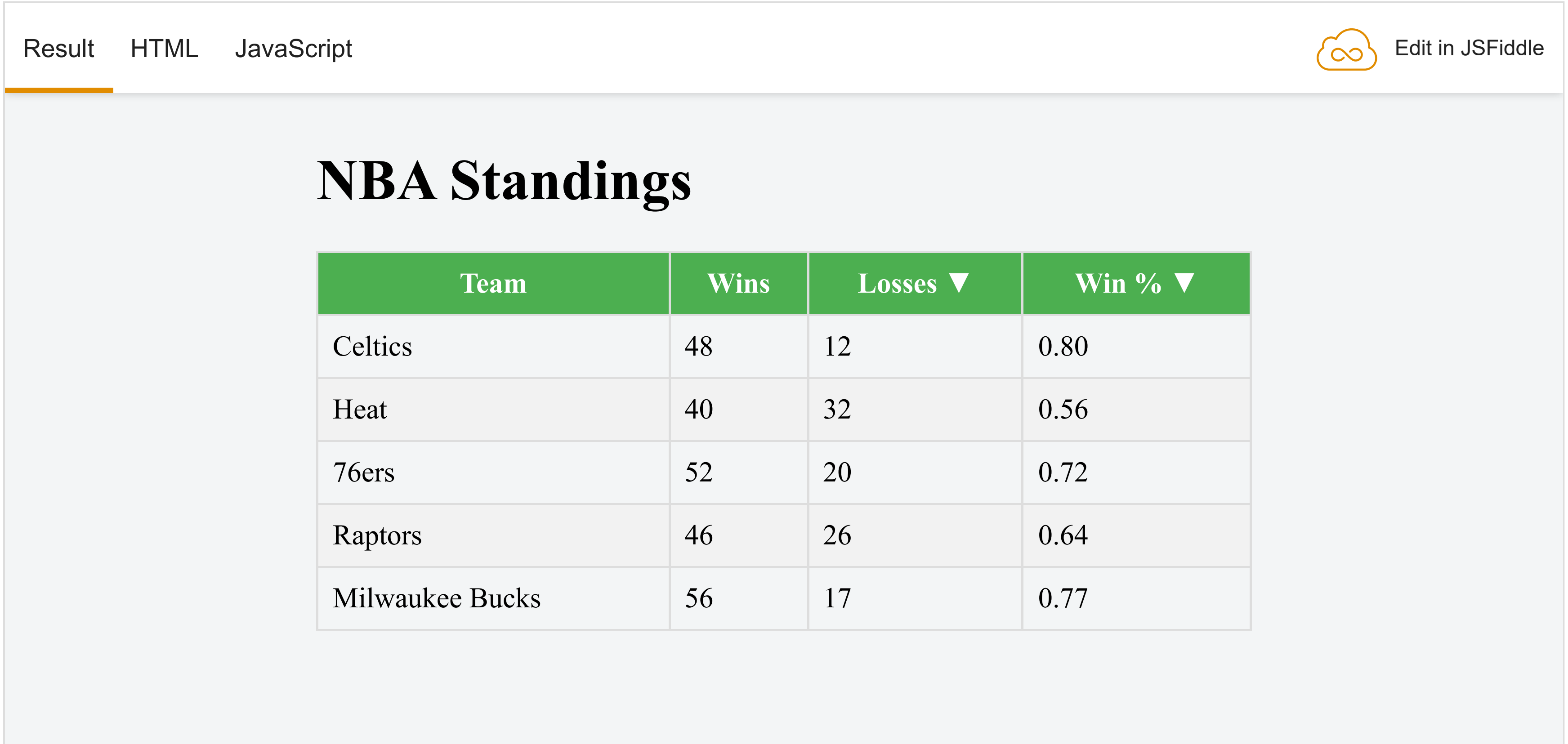
All Broadcas

**Thursday, February 24**

| Time | Teams | Broadcaster |
|---|---|---|
| 3:00 AM AST | Cleveland Cavaliers @ Detroit Pistons | NBA League Pass |
| 3:30 AM AST | Boston Celtics @ Brooklyn Nets | beIN Sports |
| 4:00 AM AST | Atlanta Hawks @ Chicago Bulls | NBA TV |

## Example 2: Sorting a Table in JS

- The following example demonstrates how to use mouse events to arrange table columns in both ascending and descending order.



### NBA Standings

| Team | Wins | Losses ▼ | Win % ▼ |
|---|---|---|---|
| Celtics | 48 | 12 | 0.80 |
| Heat | 40 | 32 | 0.56 |
| 76ers | 52 | 20 | 0.72 |
| Raptors | 46 | 26 | 0.64 |
| Milwaukee Bucks | 56 | 17 | 0.77 |

## Cookies 🍪

- Web applications can store data locally on the user's computer through the web browser.
- Cookies are an old client-side storage mechanism used to store small pieces of data on the user's computer.
- It can be used to store user preferences, identifiers for tracking user behavior, and data for analytics collection.
- Cookies are stored separately for each website (*origin*), so one website cannot access the cookies of another website.
  - *origin* refers to the combination of the protocol (e.g., HTTP or HTTPS), domain (e.g., example.com), and port (e.g., 80 or 443) of a URL. Example: `https://www.example.com:8080/login`
- Cookies in JavaScript are not secure and do not use cryptography, but transmitting them over HTTPS helps.
- While cookies are useful for storing small pieces of data, modern alternatives like local storage or IndexedDB offer better storage options.

## Setting Cookies (🍪 🍪)

- Cookies are created using the `document.cookie` property in JavaScript.

```
document.cookie = "username=Fatima";
document.cookie = "lastVisit=" + new Date().toUTCString();
```

- They are stored as key-value pairs.
- They can be accessed and modified using JavaScript
- Cookies may have an expiry date or max age.
  - `expires=date-in-UTCString-format`: The expiry date of the cookie.
  - `max-age=max-age-in-seconds`: The maximum age of the cookie in seconds.
- If neither `expires` nor `max-age` is specified, it will be deleted at the end of the session (closing a Tab or the Browser).

## Setting Cookies Example (🍪 🍪 🍪)

- The `document.cookie` property is used to set a string of all cookies.
- All cookies are stored in a single string, separated by semicolons.
- Calling `document.cookie` multiple times does not override previous cookie values.
- Each call to `document.cookie` adds a new cookie or updates an existing cookie with the same name.

```javascript
// setting a cookie without expiry date or max-age
document.cookie = "background=dark";
// setting a cookie with max-age of 2 days in seconds
document.cookie = "lang=ar; max-age=" + 2 * 24 * 60 * 60;
// setting a cookie that expires in 2 days
document.cookie = "profileId=1dsf32f39; expires=" + new Date(Date.now() + 2 * 24 * 60 * 60 * 1000).toUTCString();

console.log(document.cookie);
```

## Reading Cookies Example (🍪 🍪 🍪 🍪)

- `document.cookie` returns a string of all cookies.
- To access a cookie, we need to treat it as a string and split the key value pairs by the `;` separator.

```
// accessing the previous cookie "background=dark; lang=ar; profileId=1dsf32f39"
let myCookies = document.cookie.split(";");
let backgroundColor = myCookies[0].split("=")[1];
let language = myCookies[1].split("=")[1];
let profileId = myCookies[2].split("=")[1];
console.log(backgroundColor, language, profileId);
```

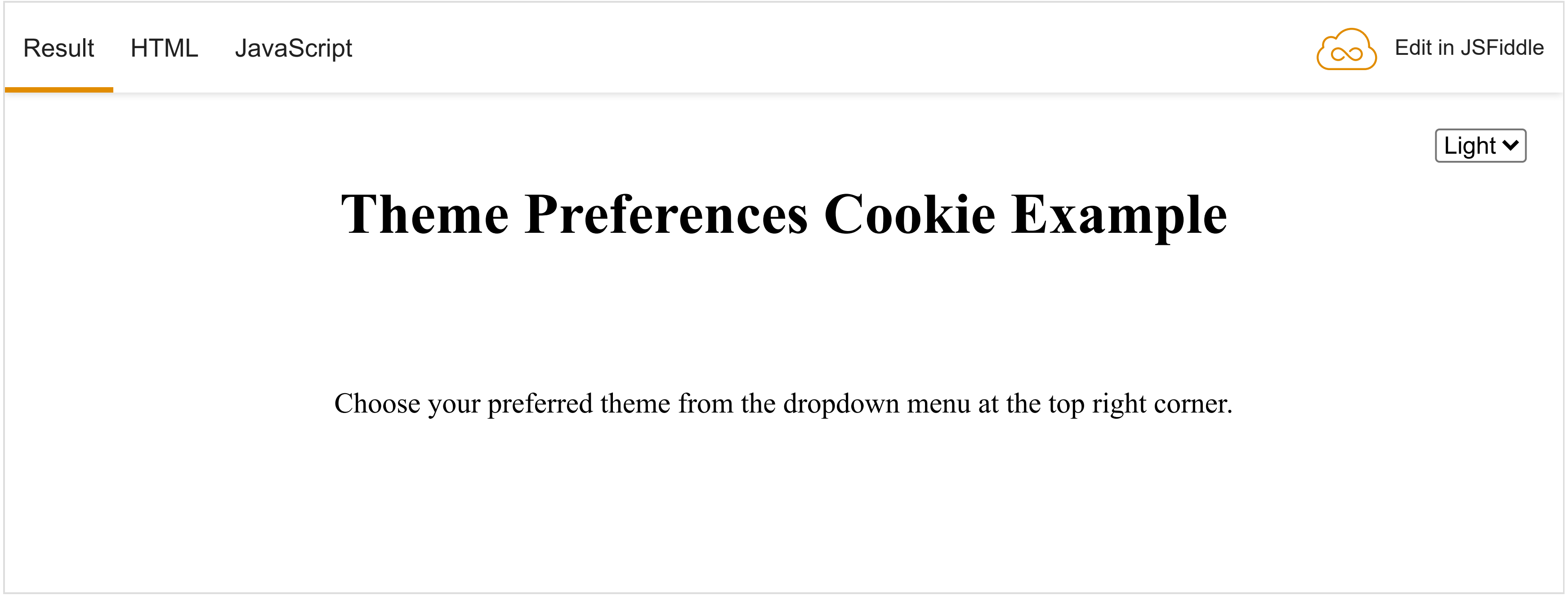
## Deleting Cookies (🍪 🍪 🍪 🍪 🍪)

- To delete a cookie, set the cookie’s expiration date to a past date.

```
// Expires on 01/01/2000 (year: 2000, month: 0 (January), day: 1)
document.cookie = "background=dark; expires=" + new Date(2000, 0, 1).toUTCString() + "; path=/";

// reading the cookie to confirm deletion
console.log(document.cookie);
```

## Cookies Example (🍪🍪🍪🍪🍪🍪)

- Using cookies to store user theme preferences (dark or light mode).



Theme Preferences Cookie Example

Choose your preferred theme from the dropdown menu at the top right corner.

- Note: JSFiddle cookies expire after reloading the page, but you can test it on GitHub Pages here.

## Part 3: The Document Object Model (DOM API)

## The Document Object Model (DOM)

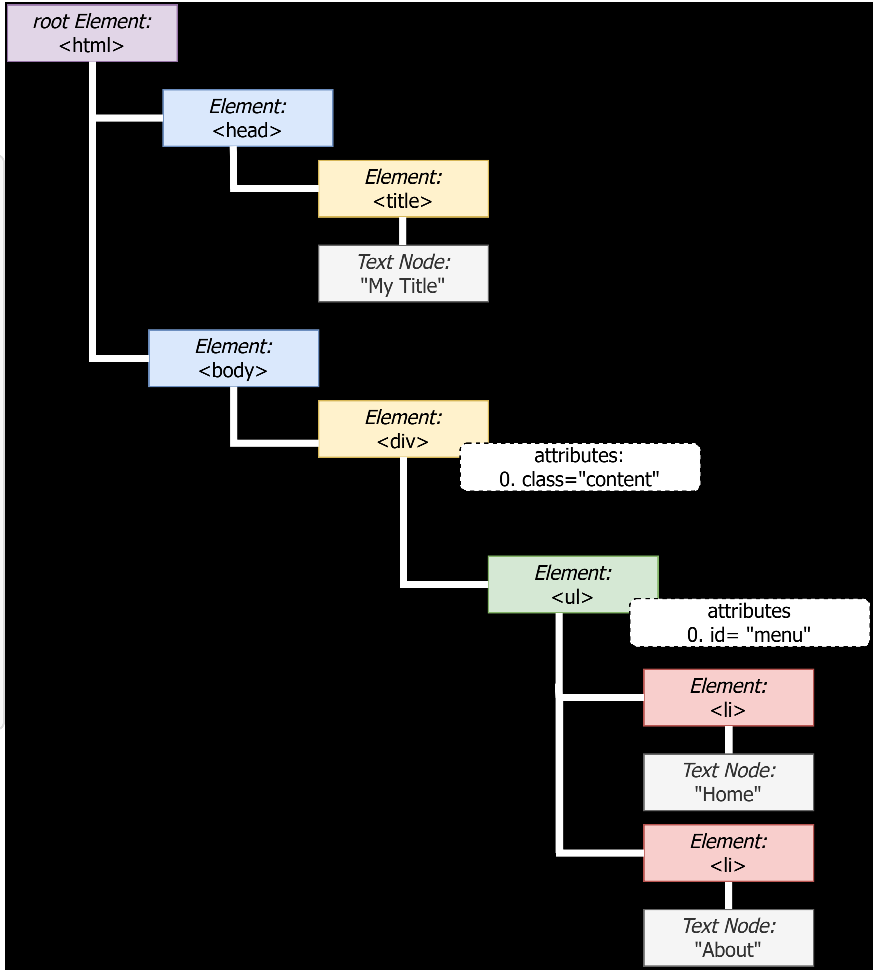
- The Document Object Model (DOM) is an application programming interface (API) for web documents.
- It represents the structure of a web page and allows programs to manipulate the content, structure, and styles of the web page.
- The DOM API enables us to create dynamic and interactive web pages by allowing JavaScript to interact with the HTML and CSS of the web page.
- Using the DOM API, we can add, change, or delete any HTML elements and attributes, CSS styles, or events dynamically.
- We will explore the main parts of the DOM API that enable these capabilities.

## Document Object Model (DOM)

- JavaScript allows us to create dynamic interactive web applications.
- This is due to the Document Object Model (DOM) interface that represents the HTML document as a logical tree of nodes.
- Each branch of the HTML tree ends in a node, and each node contains objects.
- The HTML DOM API provides access to HTML elements and their attributes, content, and CSS styles.
- This means we can change the structure, style or content of any HTML document in JavaScript.
- An object in the DOM is called a `node`.
- The DOM has some built-in nodes such as `document` and `document.body` and HTML element nodes such as `<div>` or `<p>`.
- A node can also be a `textNode`, which refers to a text contained in an element such as `<p>some text</p>` or a comment node for comments such as `<!- - comment - - >`.

## Example of DOM hierarchy in an HTML document

```
<!DOCTYPE html>
<html>
<head>
    <title>My Title</title>
</head>
<body>
    <div class="content">
        <ul id="menu">
            <li>Home</li>
            <li>About</li>
        </ul>
    </div>
</body>
</html>
```



## DOM Node Types

| Node Type | Example |
|---|---|
| `ELEMENT_NODE` | Any HTML element such as `<p>`, `<ul>`, `<div>`, etc. |
| `ATTRIBUTE_NODE` | Any attribute of an Element such as `href` in `<a>` or `src` in `img` |
| `TEXT_NODE` | The actual text inside an Element or Attribute such as `<p>text</p>` or `<img src="./some-image.png">` |
| `COMMENT_NODE` | A Comment node as in `<!-- this is a comment -->` |
| `DOCUMENT_NODE` | A Document node that represents any web page loaded in the browser |

## The `window` object in the DOM

- The `window` object is the root of the DOM
- The `window` object represents a window containing a DOM `document`.
- The `document` object, which represents the actual DOM in the current web page, is a property of the window object.
- Each tab or window in a web browser is represented by its own `Window` object.
- Each `window` object has a `document` property that refers to the actual DOM, which represents the content of the window.

## Selecting individual elements

- We can select or query elements using:
    1. Element Id values `document.getElementById(id)`
    2. Element tag names `document.getElementsByTagName(name)`
    3. Element names `document.getElementsByName(name)`
    4. CSS selectors `document.querySelector(selectors)` and `document.querySelectorAll(selectors)`

## 1. Using `document.getElementById`

- The Document method `document.getElementById(id)` returns an Element that matches the given id value.

```html
<h1 id="title">Title 1</h1>
<h1>Title 2</h1>
<p id="article">content</p>

<script>
  let elem = document.getElementById("title");
  console.log(elem.innerText);
</script>
```

```
Title 1
```

## 2. Using `document.getElementsByTagName`

- The Document method `document.getElementsByTagName(name)` returns a `NodeList` of Elements that match the given Element tag name (e.g., `p`, `h1`, `table`, etc.).
- Notice the s in the method name, getElementsByName, so you should expect a collection of elements returned not a single element.

```
<h1 id="title">Title 1</h1>
<h1>Title 2</h1>
<p id="article">content</p>

<script>
    let elems = document.getElementsByTagName("h1");
    for (let i = 0; i < elems.length; i++) {
        console.log(elems[i].innerText);
    }
</script>
```

```
Title 1
Title 2
```

## 3. Using `document.getElementsByName`

- The Document method `document.getElementsByName(name)` returns a `NodeList` of Elements that match the given name attribute, which is often used in input forms (e.g., `name="hobby"`).

```
<input name="hobby" type="checkbox" id="volleyball" value="Volleyball">
<label for="volleyball">Volleyball</label>
<input name="hobby" type="checkbox" id="football" value="Football">
<label for="football">Football</label>
<input name="hobby" type="checkbox" id="basketball" value="Basketball">
<label for="basketball">Basketball</label>

<script>
    let elems = document.getElementsByName("hobby");
    for (e of elems) {
        console.log(e.value);
    }
</script>
```

```
Volleyball
Football
Basketball
```

## 4. Using `document.querySelector()` and `document.querySelectorAll()`

- The Document methods `document.querySelector()` and `document.querySelectorAll()` allow us to find elements if they match the given CSS selector.
  - `document.querySelector()` returns the first Element that matches the specified CSS selector.
  - `document.querySelectorAll()` returns a `NodeList` of all Elements that match the specified CSS selector.

```
<h1 class="title">KAU</h1>
<h1 class="title">CPIT-405</h1>

<script>
  let firstElem = document.querySelector('h1.title');
  console.log(firstElem.textContent);

  let allElems = document.querySelectorAll('h1.title');
  allElems.forEach(elem => console.log(elem.textContent));

</script>
```

```
KAU
KAU
CPIT-405
```

## More on `querySelector` and `querySelectorAll`

- **Note:** Many browsers won't match for the psudeo-selector `:visited` and the psudeo-elements `::first-line` when using `document.querySelector()` or `document.querySelectorAll()`
- This is due to protect the privacy of users as this may expose their browsing history.

```html
<div class="box">
    <a href="#">Link #1, in the div.</a>
    <a href="#">Link #2, in the div.</a>
    <p>p in the div.
        <a href="#">Link #3, in the p that's in the div.</a>
    </p>
</div>
<script>
    console.log("Using document.querySelector()");
    let oneElem = document.querySelector(".box > a:first-child");
    console.log(oneElem.innerText);
    let AllElems = document.querySelectorAll(".box > a");
    console.log("Using document.querySelectorAll()");
    for (e of AllElems) {
        console.log(e.innerText);
    }
</script>
```

```
Using document.querySelector()
Link #1, in the div.
Using document.querySelectorAll()
Link #1, in the div.
Link #2, in the div.
```

# Changing the style of an element (I)

- We can use the document methods above to get elements and change their style in the form of `element.style.<cssProperty>`, where the CSS property's name is specified in a camelCase form
- As a general rule of thumb, in order to get the style property name in JavaScript, you should change any hyphenated CSS property's name to camelCase.

| css property | JavaScript property |
|---|---|
| background-color | backgroundColor |
| margin-top | marginTop |
| text-decoration | textDecoration |
| color | color |

## Changing the style of an element (II)

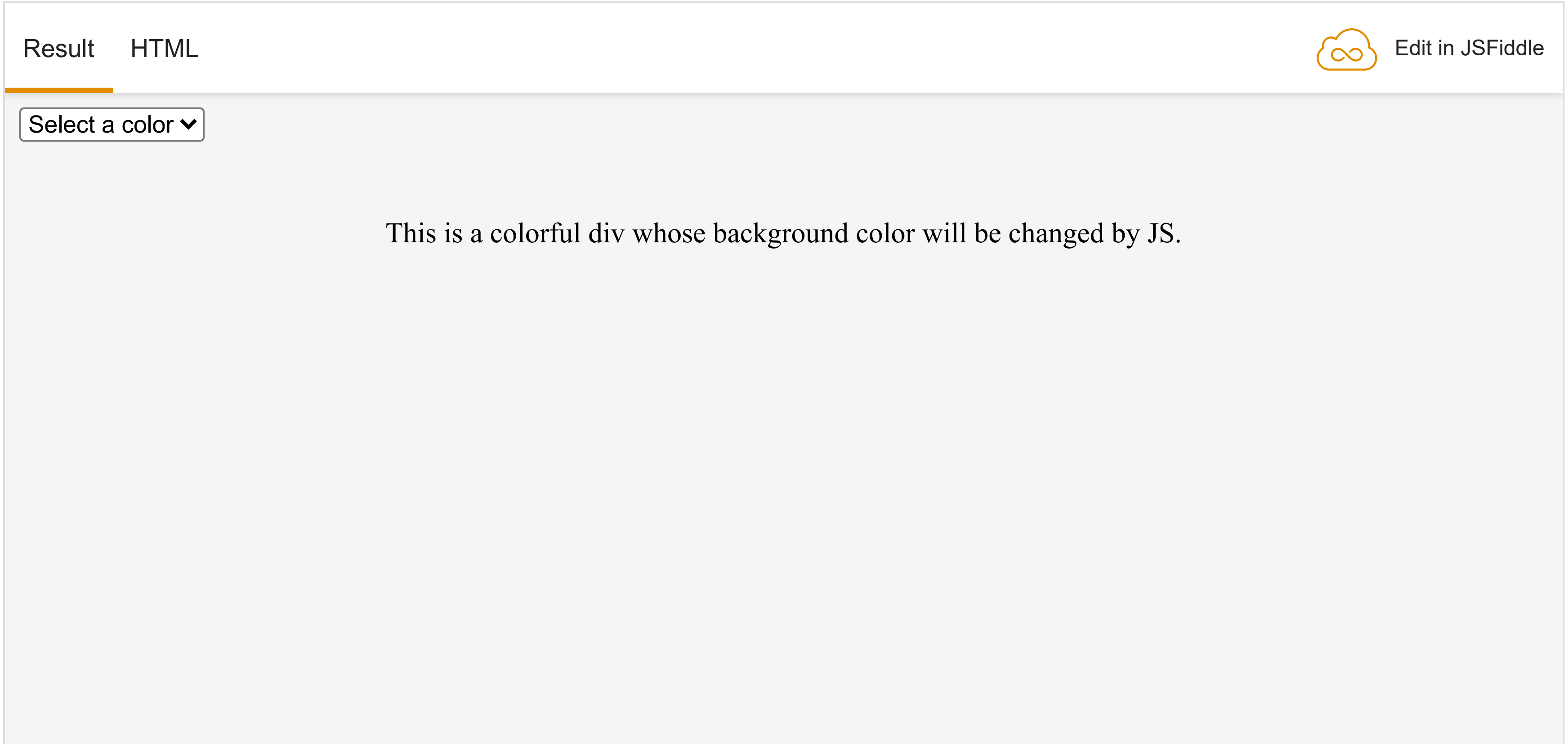
- Example on how to change the background of a `<div>` element in JavaScript:

```html
<div>
  <p>This is a paragraph</p>
</div>
```

```javascript
document.getElementsByTagName("div")[0].style.backgroundColor = "red";
```

## Changing the style of an element (III)

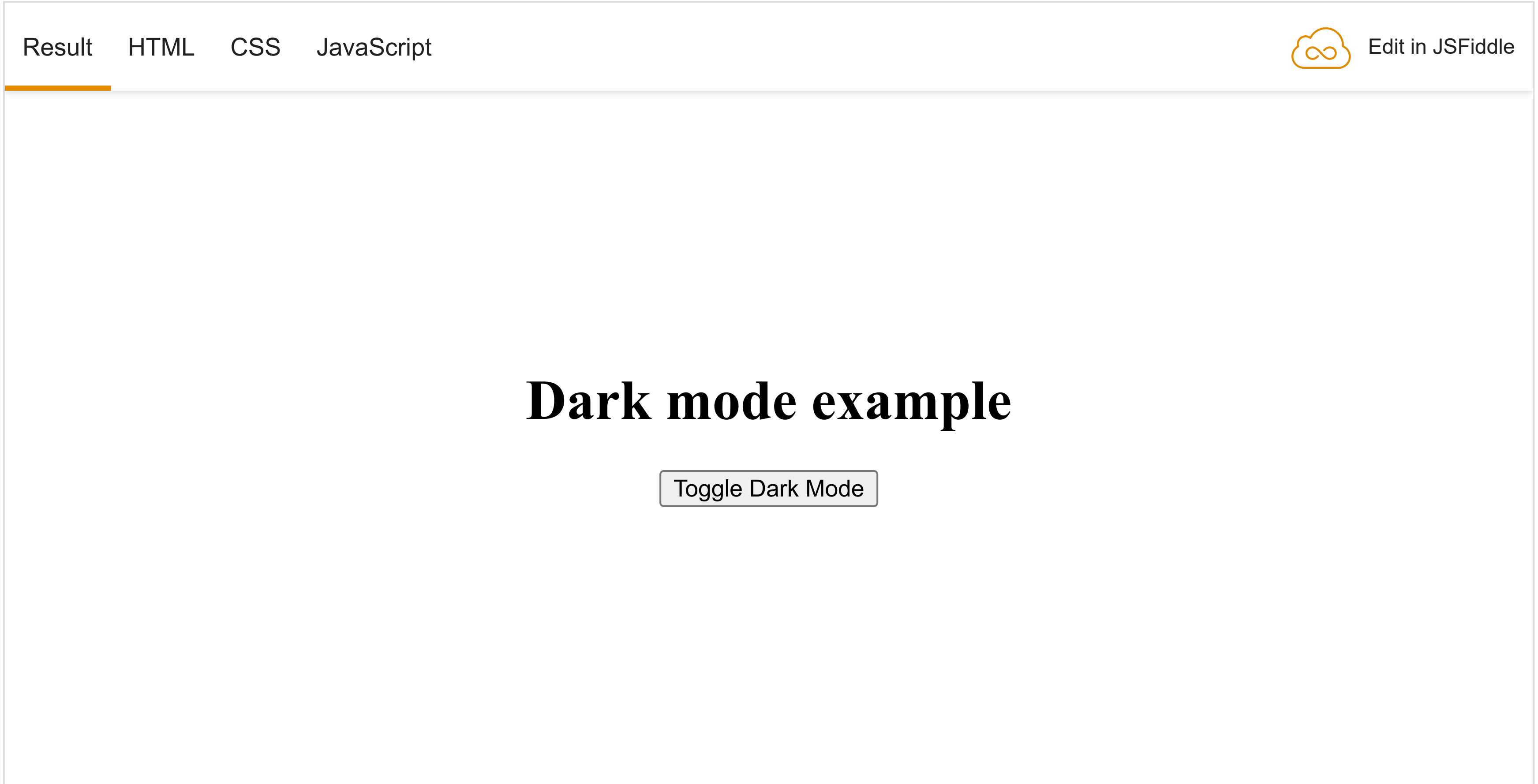
- Example on how to change the CSS background property in response to the event of selecting a color from a dropdown list.



This is a colorful div whose background color will be changed by JS.

## Changing the style of an element (IV)

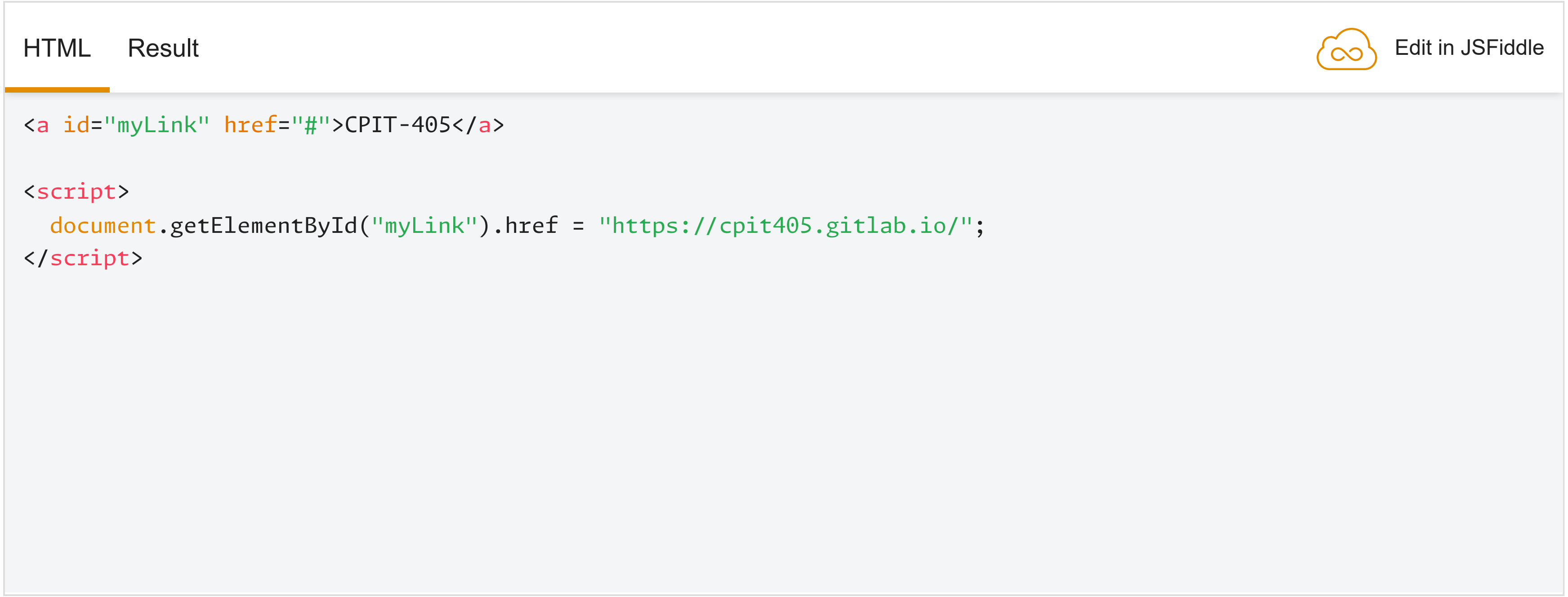
- Example on how to toggle dark mode using `classList.toggle`



Dark mode example

## Changing HTML attributes of any element

- We can use the document methods above to query or get elements and change their attributes in the form of `element.<attribute_name>` as shown below:



```html
<a id="myLink" href="#">CPIT-405</a>

<script>
  document.getElementById("myLink").href = "https://cpit405.gitlab.io/";
</script>
```

## Changing the `innerText` or `innerHTML` of an element

- We can get or set the element's **text content** using `element.innerText`.
- We can get or set the element's **HTML code** using `element.innerHTML`.

HTML Result Edit in JSFiddle

```
<a id="myLink" href="https://cpit405.gitlab.io/"></a>
<div id="codeBlock"></div>
<script>
  document.getElementById("myLink").innerText = "CPIT-405 inserted using innerText";
  document.getElementById("codeBlock").innerHTML =
    `<h3>HTML generated in JS using innerHTML</h3>
    <ul id="list">
      <li><a href="#">Item 1</a></li>
      <li><a href="#">Item 2</a></li>
      <li><a href="#">Item 3</a></li>
    </ul>`;
```

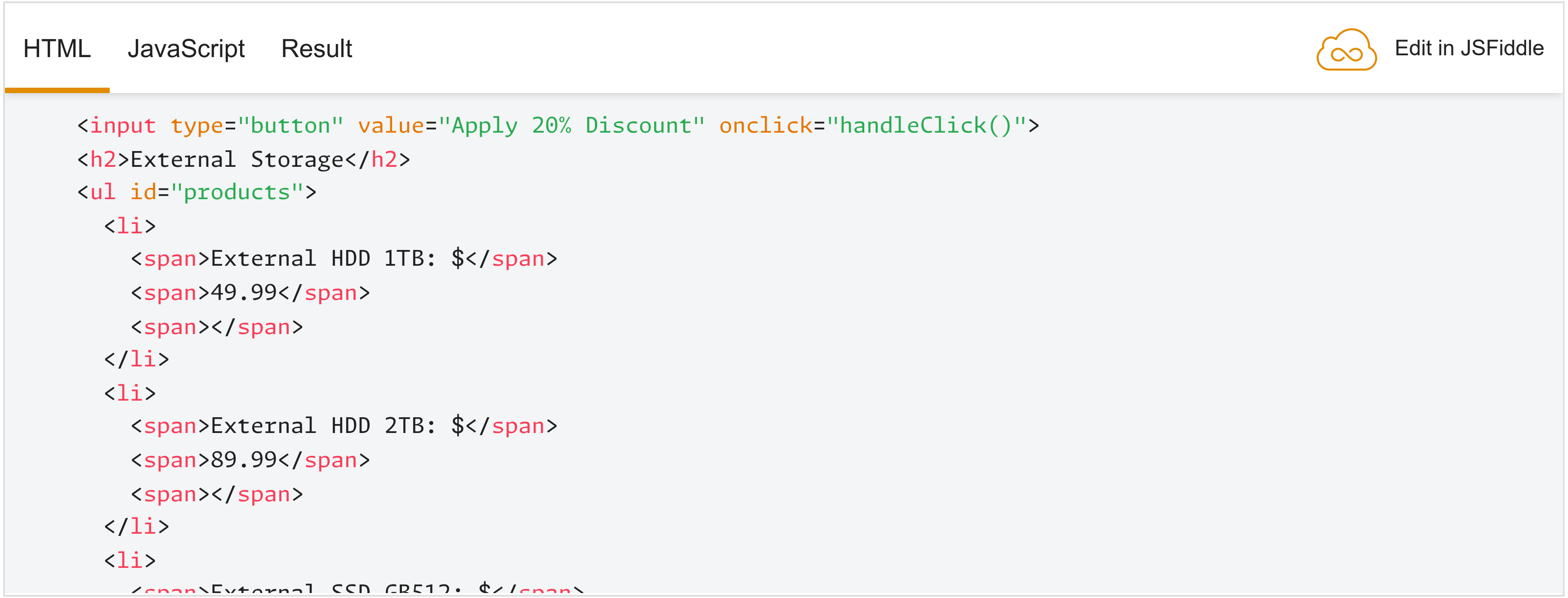
## DOM tree traversal (I)

### Getting ancestors, descendants, and siblings of an element

- Once an element is selected (e.g., using `document.getElementById()` or any other DOM method), we can traverse the element and its children using the following properties:
- **Ancestors**: get the parent, grandparent or great-grandparent, and so on as a node or element node.
  - `.parentNode`: returns the parent **node object**. A node object can be any node in the DOM such as an element node, a text node, or a comment node.
  - `.parentElement`: returns the parent node that is an **element node**.
  - `.closest(selector)`: returns the closest ancestor that matches the specified selector.
- **Descendants**: get the children, first child, and last child as a node or element node.
  - `childNodes`, `.children`, `.firstChild`, `.firstElementChild`, `.lastChild`, and `.lastElementChild`
  - **Siblings**: Get the siblings as nodes or element nodes.
    - `previousSibling`, `previousElementSibling`, `nextSibling`, and `nextElementSibling`

## DOM tree traversal (II): Example 1

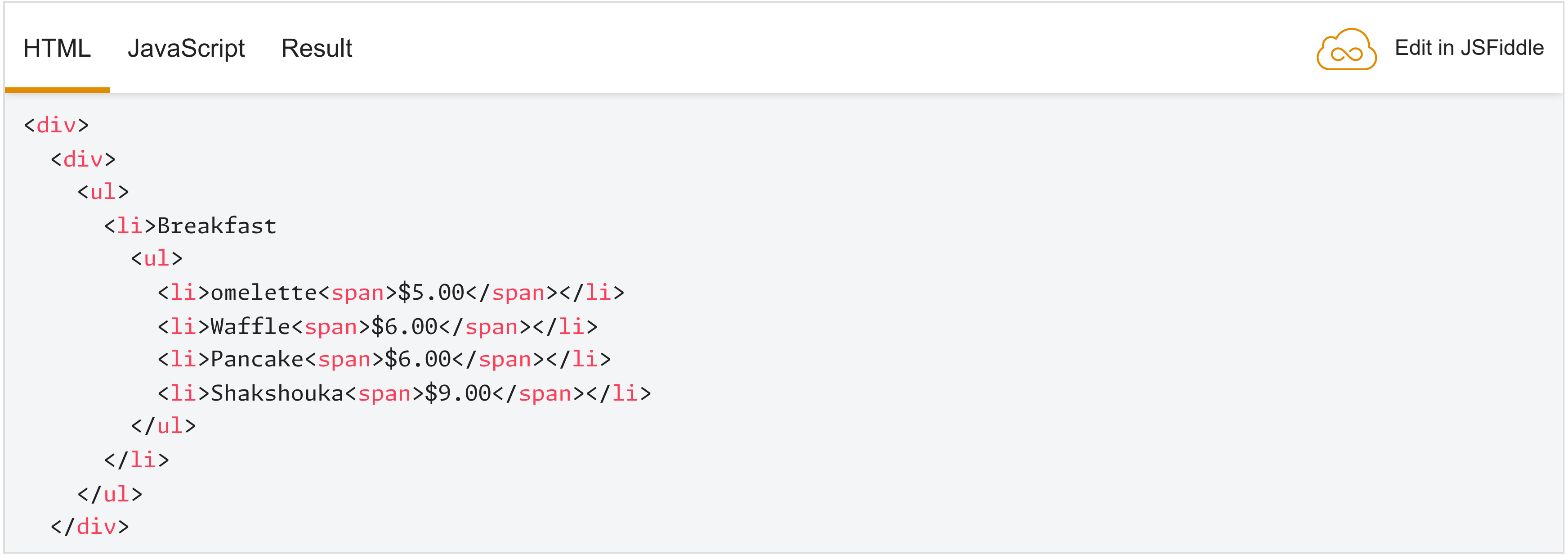
- Below is an example that shows how to traverse the DOM to apply a 20% discount to the prices of a list of items.



```html
<input type="button" value="Apply 20% Discount" onclick="handleClick()">
<h2>External Storage</h2>
<ul id="products">
  <li>
    <span>External HDD 1TB: $</span>
    <span>49.99</span>
    <span></span>
  </li>
  <li>
    <span>External HDD 2TB: $</span>
    <span>89.99</span>
    <span></span>
  </li>
  <li>
    <span>External SSD GB512: $</span>
```

# DOM tree traversal (III): Example 2

- We can use the element’s `.closest()` method to traverse the Element and its parents (moving up the DOM tree toward the document root) until it finds a node that matches the given selector string.
- Suppose we want to style the closest `ul` element to a list of items on a page that has multiple nested `ul` elements:



```html
<div>
  <div>
    <ul>
      <li>Breakfast
        <ul>
          <li>omelette<span>$5.00</span></li>
          <li>Waffle<span>$6.00</span></li>
          <li>Pancake<span>$6.00</span></li>
          <li>Shakshouka<span>$9.00</span></li>
        </ul>
      </li>
    </ul>
  </div>
```

## Creating, inserting, and deleting nodes
## Using `createElement`, `createTextNode`, and `appendChild`

- We can use the following methods:
  - `document.createElement`: create an HTML element
  - `document.createTextNode`: create a text node for the text value inside of an element.
  - `<Element>.appendChild`: add a node to the end of the list of children of a parent node.

JavaScript Result Edit in JSFiddle

```javascript
// create <ul><li>Item 1</li><li>Item 2</li></ul>
var ulElem = document.createElement("ul");

var liElem1 = document.createElement("li");
var textNode1 = document.createTextNode("Item 1");
liElem1.appendChild(textNode1);

var liElem2 = document.createElement("li");
var textNode2 = document.createTextNode("Item 2");
liElem2.appendChild(textNode2);

// append the li child elements to the ul element
```

## Inserting text before or after an element using `insertAdjacentText` (I)

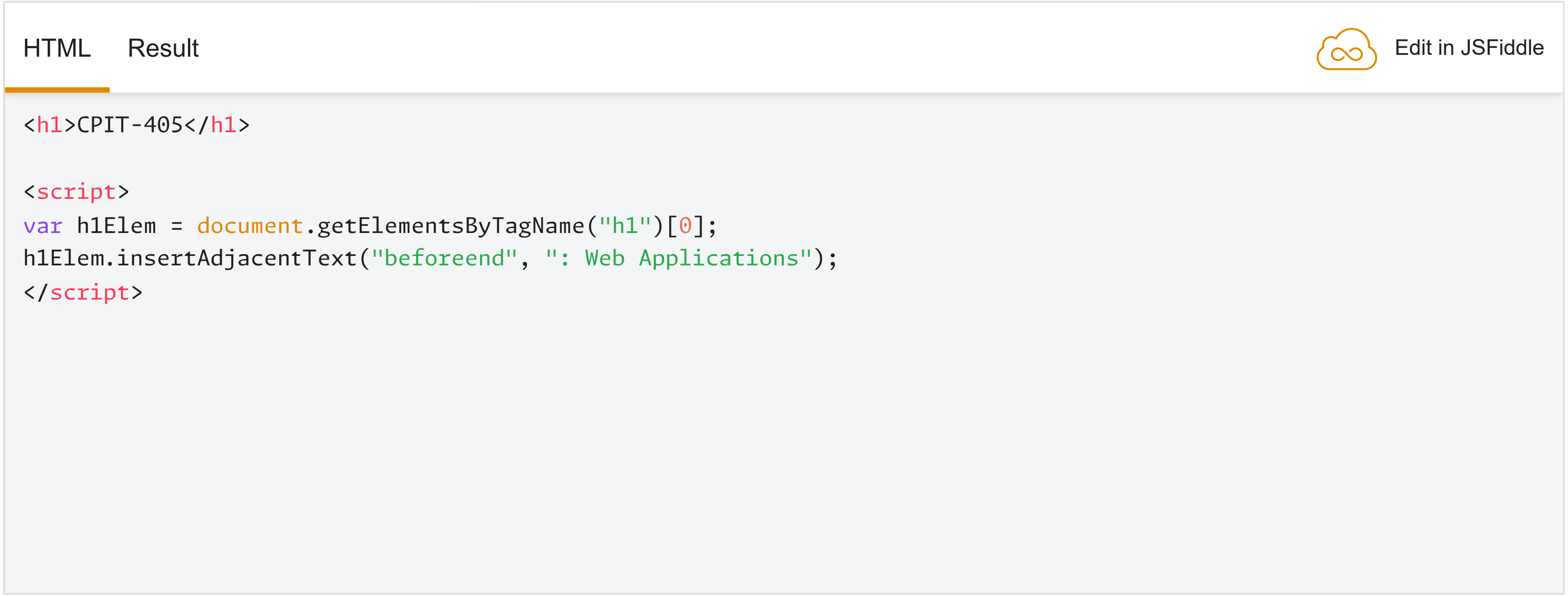
- We can use `insertAdjacentText(where, text)` to insert text before an element, at the beginning of the element’s text, at the end of the element’s text, or after the element itself.
  - Example: `myElement.insertAdjacentText("beforeend", ": Web Applications");`

| position | Description |
|---|---|
| `beforebegin` | Before the element itself. |
| `afterbegin` | Inside the element but before its first child. |
| `beforeend` | Inside the element but ater its last child. |
| `afterend` | After the element itself. |

- Example of the possible positions

```
<!-- beforebegin -->
<p>
  <!-- afterbegin -->
  text value
  <!-- beforeend -->
</p>
<!-- afterend -->
```

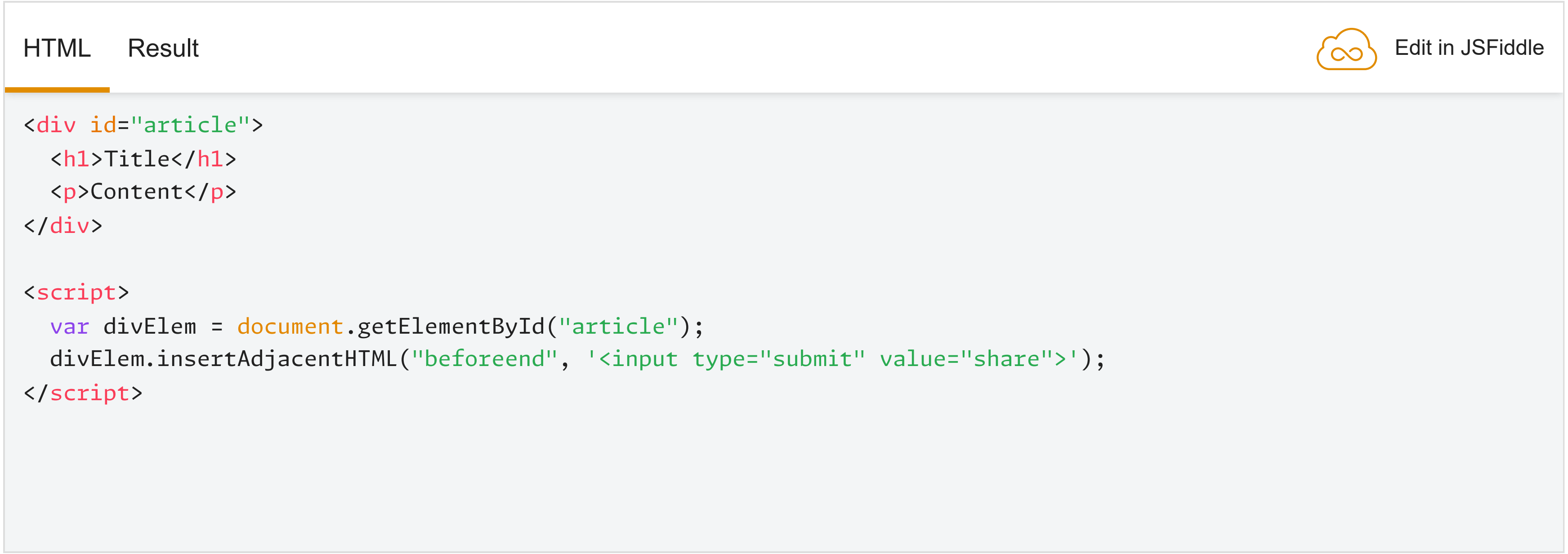
## Inserting text before or after an element using `insertAdjacentText` (II): Example



```
<h1>CPIT-405</h1>

<script>
var h1Elem = document.getElementsByTagName("h1")[0];
h1Elem.insertAdjacentText("beforeend", ": Web Applications");
</script>
```

## Inserting HTML before or after an element using `insertAdjacentHTML`

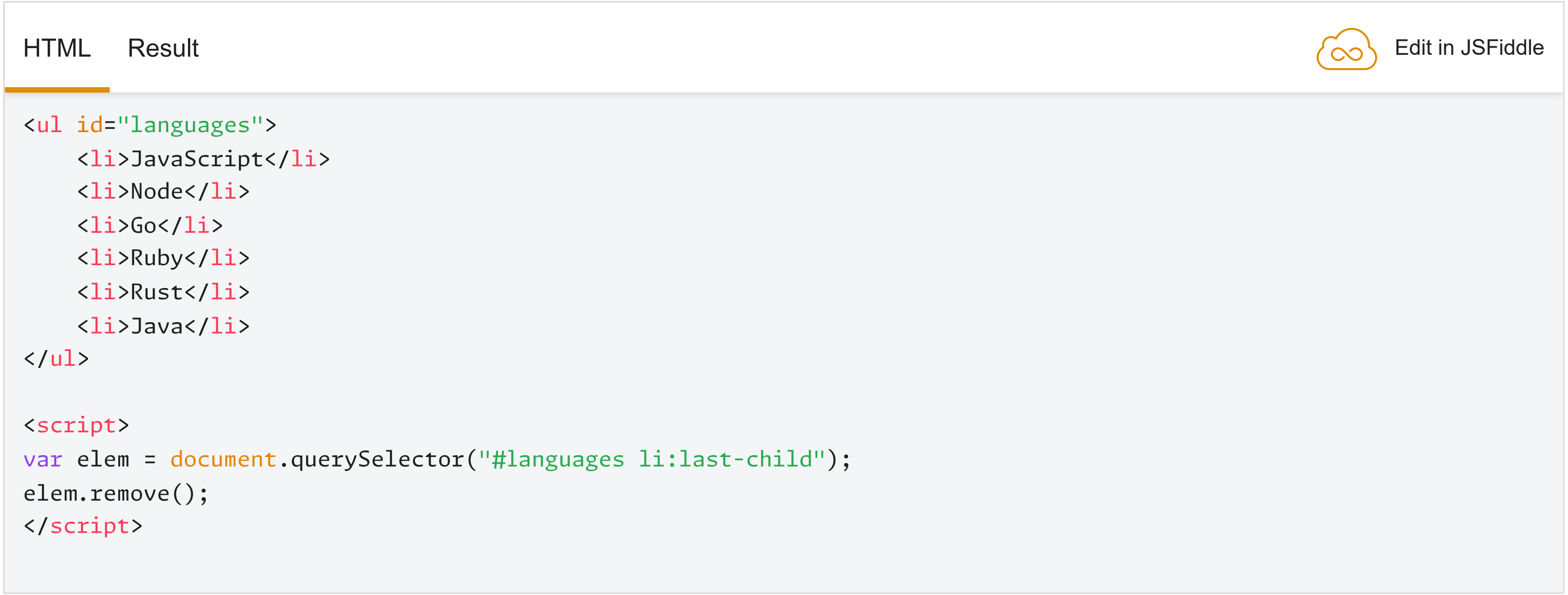
- The `insertAdjacentHTML(where, text)` is like the `insertAdjacentText` but with inserting HTML as a string instead of text.



```
<div id="article">
  <h1>Title</h1>
  <p>Content</p>
</div>

<script>
  var divElem = document.getElementById("article");
  divElem.insertAdjacentHTML("beforeend", '<input type="submit" value="share">');
</script>
```

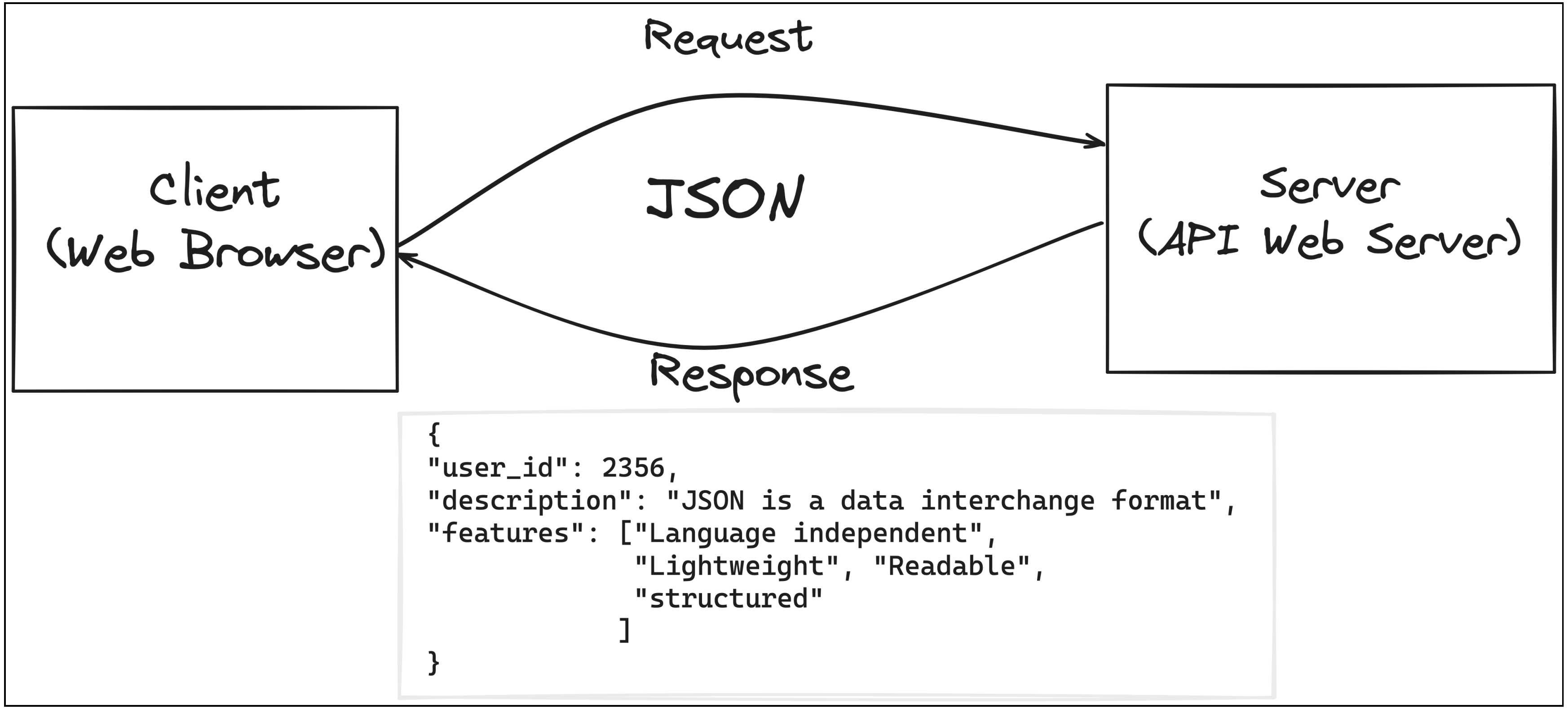
## Deleting an element using `.remove()`

- We can use the element's `.remove()` method to remove elements from the DOM tree.



```
<ul id="languages">
    <li>JavaScript</li>
    <li>Node</li>
    <li>Go</li>
    <li>Ruby</li>
    <li>Rust</li>
    <li>Java</li>
</ul>

<script>
var elem = document.querySelector("#languages li:last-child");
elem.remove();
</script>
```

## Part 4: JSON



```
{
"user_id": 2356,
"description": "JSON is a data interchange format",
"features": ["Language independent",
             "Lightweight", "Readable",
             "structured"
            ]
}
```

## JSON: Introduction (II)

- JSON stands for JavaScript Object Notation
- JSON is a lightweight, text-based, language-independent data interchange format
- The goal is to facilitate structured data-interchange between the web browser and the web server.
- JSON is based on a small subset of JavaScript, but it is programming language independent
- JSON provides a simple notation that is easy to read and write
- Often used with AJAX as a replacement for XML
- Most programming languages have support for encoding and decoding JSON

## JSON: Syntax

- Uses key/value pairs (e.g., `{"name": "Ali"}` )
  - A key is a string in double quotes
  - A value can be a string in double quotes, a number, true, false, null, an object, or an array.
- Pairs are separated by commas (e.g., `{"name": "Ali", "age": 20}`)
- Curly braces hold zero or more name/value pairs
- Square brackets hold arrays
- Double quotes around both the name (key) and value
- Filename extension is **.json**
- The Internet media type (also known as MIME type), which is a standard that indicates the type and format of a document, for JSON is `application/json`.

## JSON data types

- JSON Values must be in one of the following types:
  - **Number:** JSON treats numbers as a sequence of digits. No distinction between integer and floating-point
  - **String**: A sequence of zero or more Unicode characters enclosed in double quotes
  - **Boolean:** true or false
  - **Array**: Ordered list of zero or more values wrapped in square brackets
  - **Object**: Unordered collection of key/value pairs
  - **Null:** Empty value using the word null

## JSON Example

```
{
    "name": "Ali",
    "age": 26,
    "employed": true,
    "car": null,
    "address": {
          "city": "Jeddah",
          "street": "300 Airport Rd."
    },
    "languages": [ "JavaScript", "Python", "go"]
}
```

Object

Array

## Comparison with XML

- Both JSON and XML can be used to exchange data with a server.
- JSON is often shorter, faster, and easier to read and write
- Unlike XML, comments are not supported in JSON
- JSON can be parsed by a standard JavaScript function while XML is parsed with a special XML parser.
- Most RESTful APIs (e.g., Twitter API, Instagram API, and GitHub API) return responses in JSON format only.

### JSON Example

```json
{
  "course": {
    "name": "Web Applications",
    "department": "IT",
    "code": "CPIT",
    "number": 405
  }
}
```

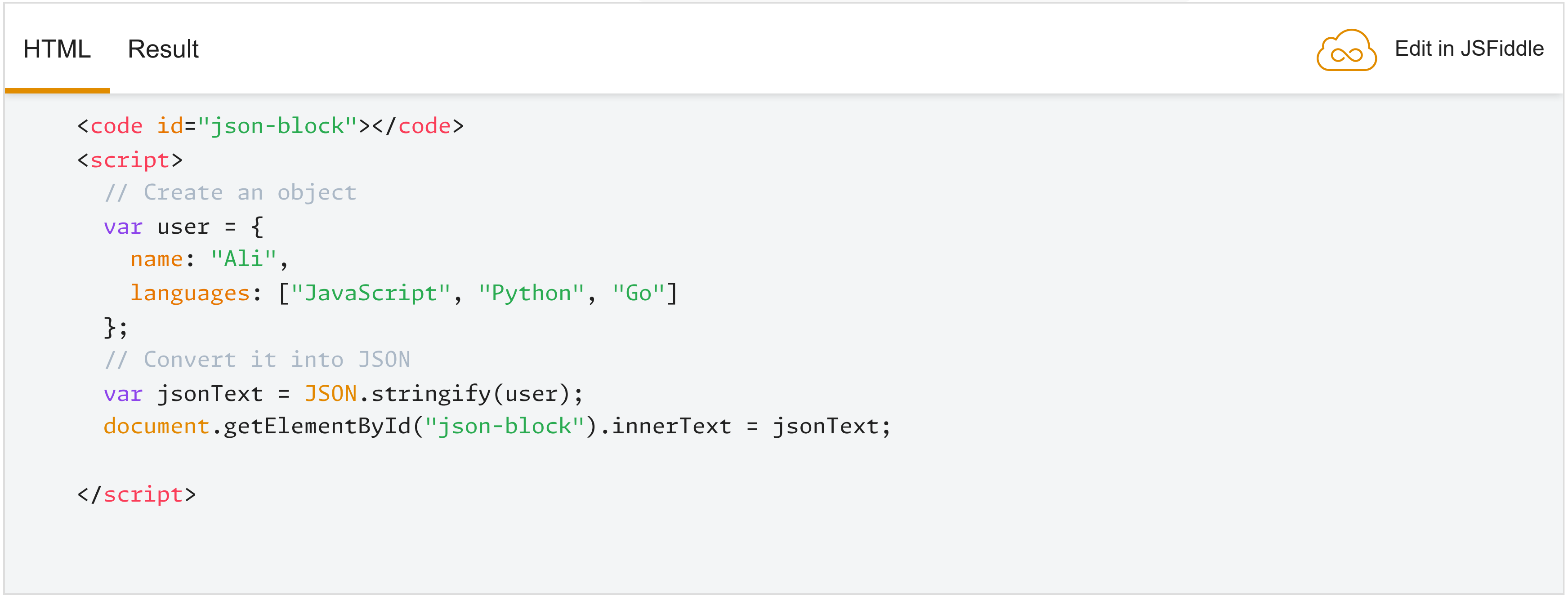
### XML Example

```xml
<!-- This is a comment in XML -->
<course>
  <name>Web Applications</name>
  <department>IT</department>
  <code>CPIT</code>
  <number>405</number>
</course>
```

## Creating and Parsing JSON

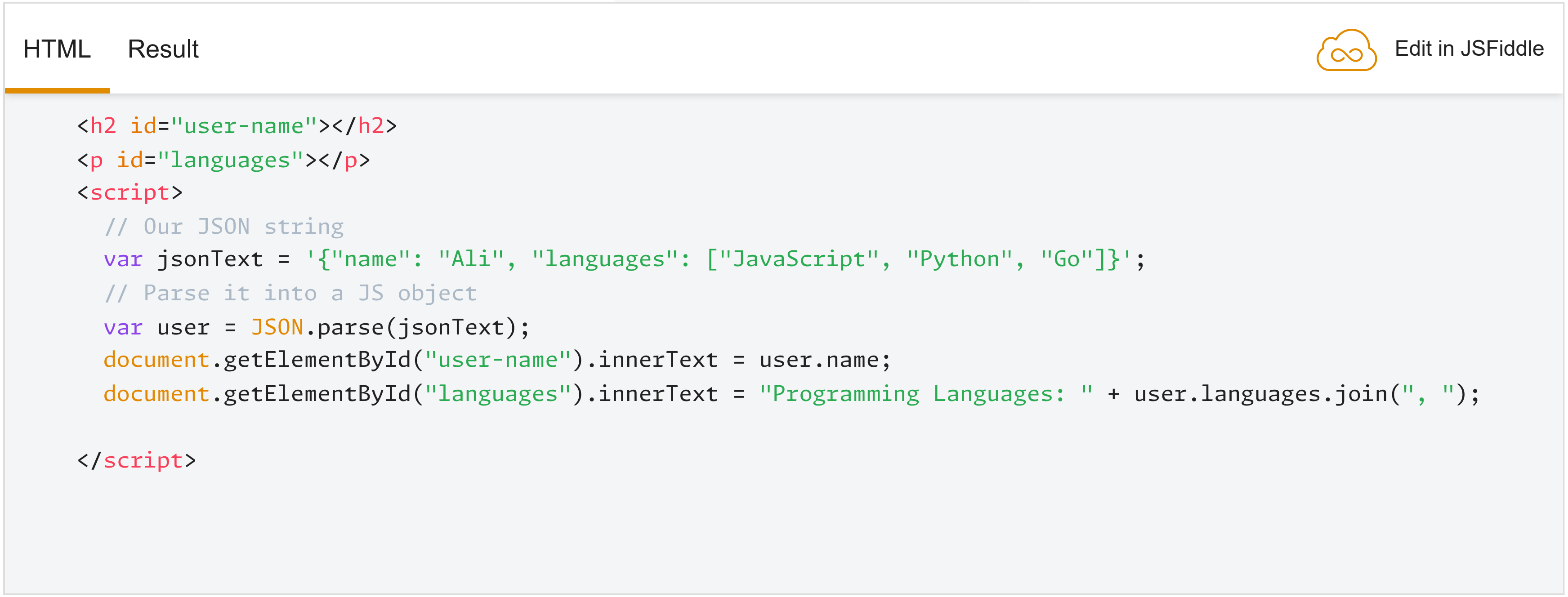
- JSON is often used to exchange data to/from a web server
- Data is always sent and received over HTTP in a string format
- To convert a JavaScript object into string, we use JSON.stringify()
- To convert a string into a JavaScript object, we use JSON.parse()

## Creating JSON in JS using `JSON.stringify(value)`



```
<code id="json-block"></code>
<script>
  // Create an object
  var user = {
    name: "Ali",
    languages: ["JavaScript", "Python", "Go"]
  };
  // Convert it into JSON
  var jsonText = JSON.stringify(user);
  document.getElementById("json-block").innerText = jsonText;

</script>
```

## Parsing JSON in JS using `JSON.parse(text)`



```html
<h2 id="user-name"></h2>
<p id="languages"></p>
<script>
  // Our JSON string
  var jsonText = '{"name": "Ali", "languages": ["JavaScript", "Python", "Go"]}';
  // Parse it into a JS object
  var user = JSON.parse(jsonText);
  document.getElementById("user-name").innerText = user.name;
  document.getElementById("languages").innerText = "Programming Languages: " + user.languages.join(", ");

</script>
```

## JSON Validation

- JSON data is often written and sent without whitespaces to save space; hence, it becomes difficult to read and solve syntax errors in JSON.
- JSON data that does not follow JSON syntax rules can’t be parsed by `JSON.parse()`
- Tools that aid in formatting and debugging JSON data is often called beautifiers and linters respectively.
- Many text editors and IDEs (e.g., VS Code and webstorm) provide tools for formatting JSON data, so that it is easy to read and debug by web developers.

## Part 5: Asynchronous JavaScript and Ajax

## Synchronous programming vs Asynchronous programming (I)

- Synchronous programming: instructions are performed one at a time and in order.
  - If a function is called, execution waits until that function returns before moving on to the next line of code.
- Asynchronous programming: instructions can be performed independently of the main program flow.
  - If a function is called, execution doesn't wait for it to complete and moves on to the next line of code.

- Synchronous example:

```
function syncFunction(a, b) {
  let sum = a + b;
  return sum
}

console.log(syncFunction(5, 5));
console.log("This is a synchronous function");
```

```
10
This is a synchronous function"
```

- Asynchronous example:

```
  function asyncFunction() {
    setTimeout(() => {
      console.log("This is an asynchronous function");
      }, 2000);
}
asyncFunction();
console.log("This is called immediately");
```

```
This is called immediately
This is an asynchronous function
```

## Synchronous programming vs Asynchronous programming (II)

- Some programs are asynchronous as they have to stop computation while waiting for network requests to complete, data to load, or for some event to occur.
  - Web applications inherently operate in an asynchronous manner.
    - Waiting for user event to occur
    - Waiting for server response to arrive

## Promises

- Promises in JavaScript are objects that represent the eventual completion or failure of an asynchronous operation, and its resulting value.

```
let promise = new Promise(function(resolve, reject) {
  // async function using setTimeout
  setTimeout(function() {
    resolve('Promise resolved')
  }, 500)
})

promise.then(function(value) {
  console.log(value)
}, function(reason) {
    console.log(reason)
})
```

## Asynchronous JavaScript and XML (Ajax)

- Asynchronous JavaScript and XML (Ajax) is an approach to using a set of existing web technologies to create asynchronous web applications that are capable of fetching resources over a network and update the web page without reloading the entire page.
- This makes web applications faster and more responsive to user actions.
- Although the X in Ajax stands for XML, JSON is preferred over XML nowadays because of its many advantages such as being more readable, lighter in size and very close to JavaScript with native support for parsing JSON using the built in method `JSON.parse`.

## Ajax in JS

- In JavaScript, there are three primary methods to retrieve or send data to an API server.

  1. **The XMLHttpRequest API**
  2. **The Fetch API**
  3. **The Fetch API with async/await**

- We will be using Postman ↗️ to send HTTP requests and inspect the responses, so we can parse and traverse the returned JSON responses easily.

## Using the `XMLHttpRequest` API

- The `XMLHttpRequest` (XHR) object is used to send HTTP requests over the network to web servers. This API allows us to send requests and retrieve data from a resource without having to do a full page reload. This API helps us create a better user experience for web applications since we can update parts of a web page without having to reload the whole page and interrupt the user while interacting with our web page.

- The `XMLHttpRequest.readyState` property returns the state an XMLHttpRequest client is in. An XHR client may be in any of the following states:

### Syntax

```
var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
  if (xhr.readyState != 4) return;
  if (xhr.status === 200) {
    console.log(xhr.responseText);
  }
};
xhr.open('GET', 'https://jsonplaceholder.typicode.com/todos/', true);

xhr.send();
```

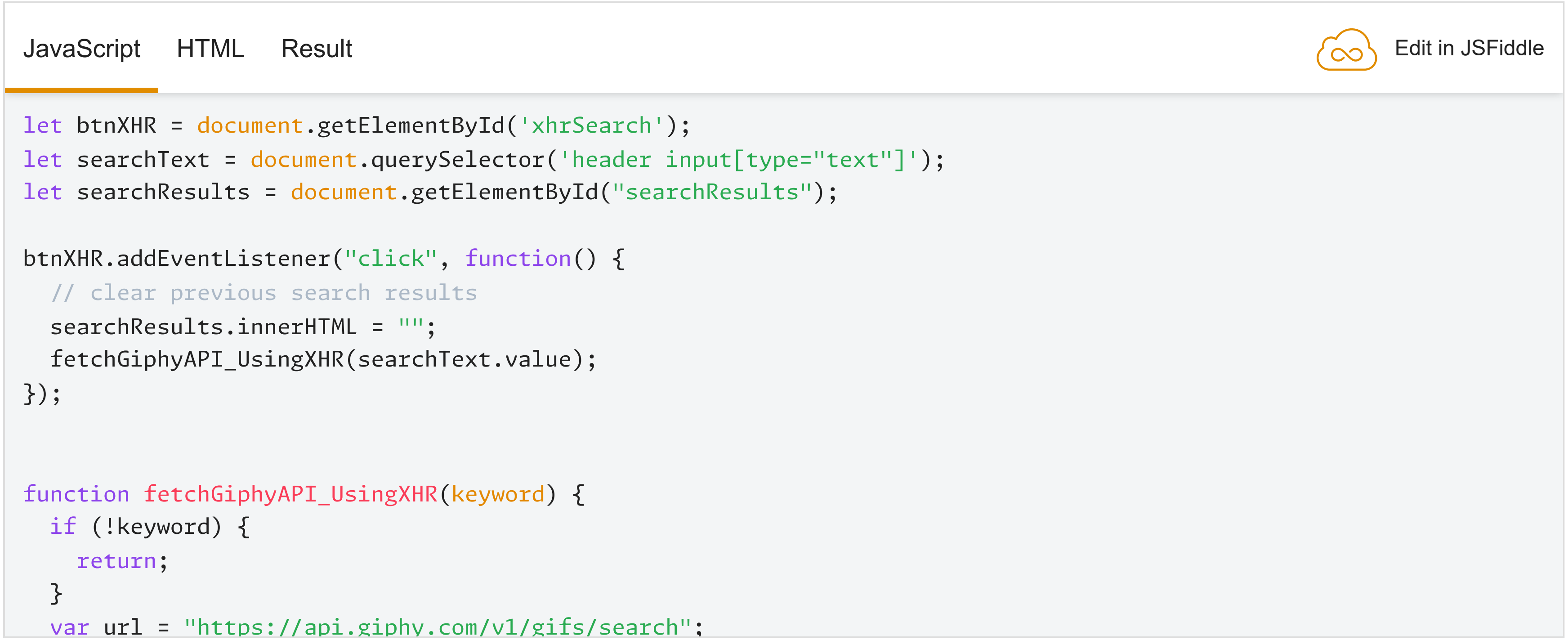
## XMLHttpRequest states `xhr.readyState`

- The `xhr.readyState` property returns the state an `XMLHttpRequest` client is in.
- An `XMLHttpRequest` client goes through several states from its creation to the completion of the HTTP request.
- The following table lists all the possible values for `xhr.readyState` :

| Value | State | Description |
|---|---|---|
| 0 | UNSENT | Client has been created but `open()` not called yet. |
| 1 | OPENED | `open()` has been called. |
| 2 | HEADERS_RECEIVED | `send()` has been called, and headers and status are available. |
| 3 | LOADING | Processing/Downloading; `responseText` holds partial data. |
| 4 | DONE | The operation is now complete. |

## `XMLHttpRequest` Example

- The following example uses XMLHttpRequest to send an HTTP GET request to the Giphy API to obtain images for the given keyword.



```javascript
let btnXHR = document.getElementById('xhrSearch');
let searchText = document.querySelector('header input[type="text"]');
let searchResults = document.getElementById("searchResults");

btnXHR.addEventListener("click", function() {
  // clear previous search results
  searchResults.innerHTML = "";
  fetchGiphyAPI_UsingXHR(searchText.value);
});


function fetchGiphyAPI_UsingXHR(keyword) {
  if (!keyword) {
    return;
  }
  var url = "https://api.giphy.com/v1/gifs/search";
```

## Using the Fetch API `fetch()` with Promises

- The `fetch` API provides an interface for making HTTP requests to access resources across the network.
- This API provides similar functionalities as the `XMLHTTPRequest`, so we can make HTTP requests (e.g., using GET, POST methods) to send data, get data, download resources, or upload files.
- The `fetch` API is much easier to use than XHR and provides more powerful and flexible feature set.

## Fetch API Syntax

- `const fetchResponsePromise = fetch(resource [, init])` Where:
  - `resource` is a string that contains the URL to the resource or a `Request` object ↗.
  - `init` is an optional object that contains custom settings for the request such as the HTTP method, headers, credentials (cookies, HTTP authentication entries), etc. [1].
- The `fetch()` method starts fetching a resource from the network and iimmediatly returns a `promise` object ↗ which is fulfilled once the response is received, which can be obtained inside the `.then()` method. If a network error is encountered, the promise is rejected and an error is thrown, which can be caught and handled inside the `.error()` method.

---

1. fetch on MDN docs ↗. ↩

## `fetch` with promises Example

- The following example uses the `fetch` API with promises to send an HTTP GET request to the Giphy API to obtain images for the given keyword.

JavaScript HTML Result Edit in JSFiddle

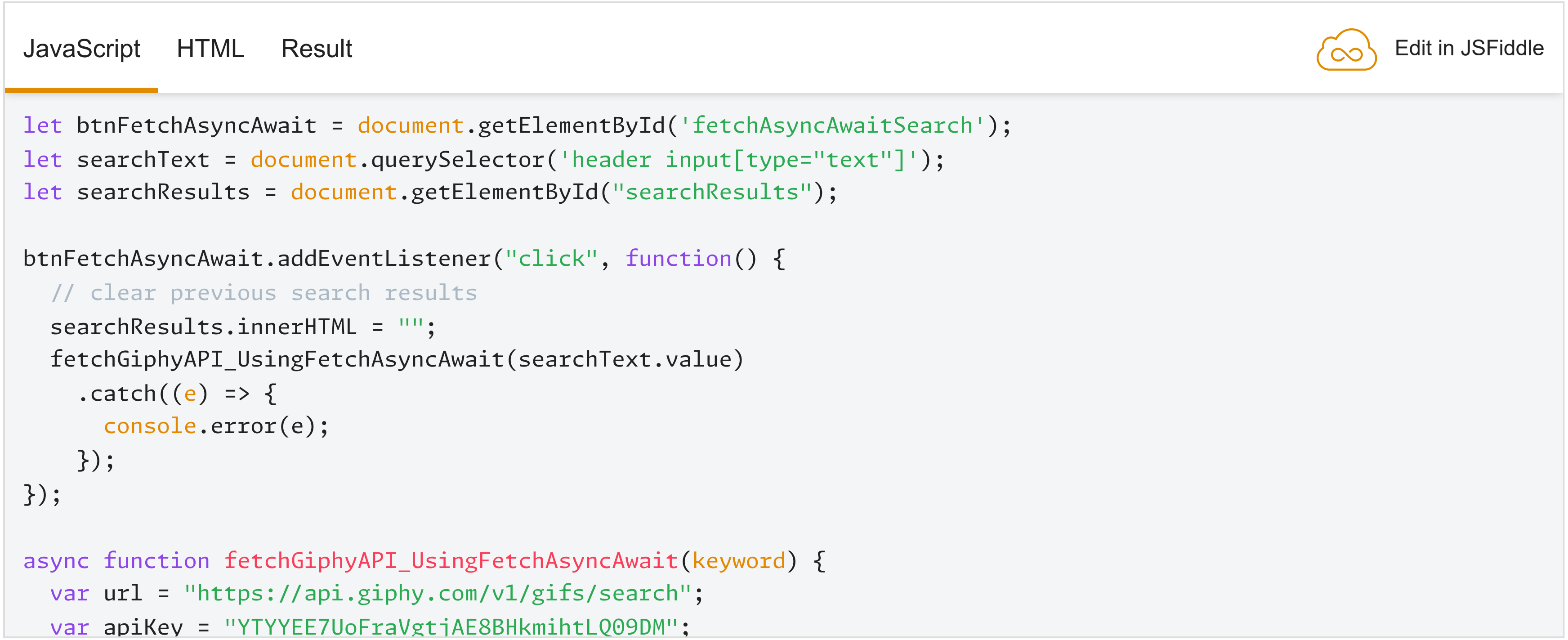
```
let btnFetch = document.getElementById('fetchSearch');
let searchText = document.querySelector('header input[type="text"]');
let searchResults = document.getElementById("searchResults");

btnFetch.addEventListener("click", function() {
  // clear previous search results
  searchResults.innerHTML = "";
  fetchGiphyAPI_UsingFetch(searchText.value);
});

function fetchGiphyAPI_UsingFetch(keyword) {
  if (!keyword) {
    return;
  }
  var url = "https://api.giphy.com/v1/gifs/search";
  var apiKey = "YTYYEE7UoFraVgtjAE8BHkmihtLQ09DM";
```

## Using the Fetch API with `async/await`

- The `async/await` syntax simplifies the writing of asynchronous code the work with promises by providing a syntax similar to that of writing synchronous code.
- This syntax is more cleaner style and helps us avoid the need to explicitly write promise chains (e.g., `.then().then().then()`).
- An `async` function is a function declared with the `async` keyword, which allows us to use the `await` keyword within the function.

## `fetch` with `async` and `await` Example

- The following example uses the `fetch` API with promises to send an HTTP GET request to the Giphy API to obtain images for the given keyword.



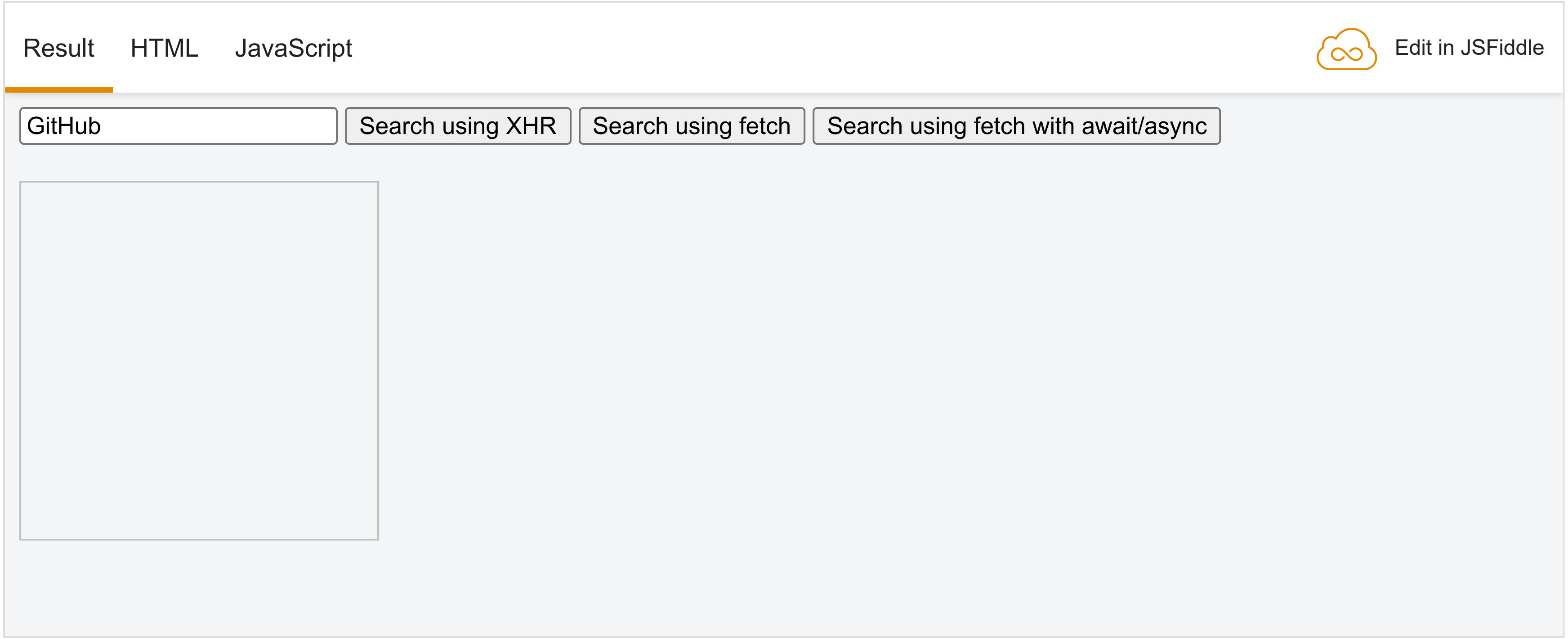
```javascript
let btnFetchAsyncAwait = document.getElementById('fetchAsyncAwaitSearch');
let searchText = document.querySelector('header input[type="text"]');
let searchResults = document.getElementById("searchResults");

btnFetchAsyncAwait.addEventListener("click", function() {
  // clear previous search results
  searchResults.innerHTML = "";
  fetchGiphyAPI_UsingFetchAsyncAwait(searchText.value)
    .catch((e) => {
      console.error(e);
    });
});

async function fetchGiphyAPI_UsingFetchAsyncAwait(keyword) {
  var url = "https://api.giphy.com/v1/gifs/search";
  var apiKey = "YTYYEE7UoFraVgtjAE8BHkmihtLQ09DM";
```

## Additional Examples

- GitHub API
- Imgur API

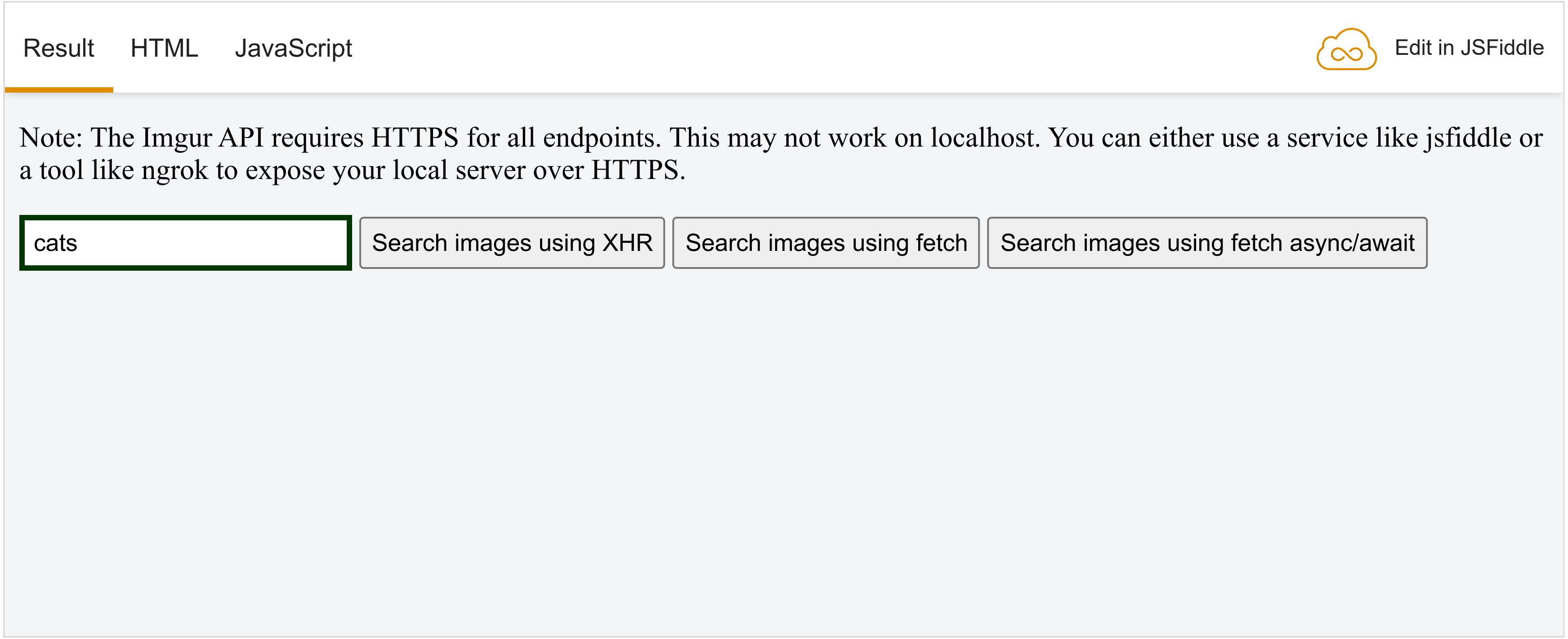
## Example: GitHub API

- Below is an example on using GitHub API ↗



## Example: Imgur API

- Below is an example on using imgur API ↗



Note: The Imgur API requires HTTPS for all endpoints. This may not work on localhost. You can either use a service like jsfiddle or a tool like ngrok to expose your local server over HTTPS.

cats
Search images using XHR
Search images using fetch
Search images using fetch async/await

## Part 6: Functional Programming in JS

## Functional Programming in JS (I)

- Functional programming is a programming paradigm where programs are constructed by applying and composing functions.
- JavaScript isn’t a functional programming language like Lisp or Haskell.
- However, its ability to treat functions as objects allows us to apply functional programming techniques.
- Functional programming is a form of declarative programming.
    - In declarative programming, you describe what you want to do without specifying how to do it.
    - In imperative programming, you explicitly describe what you do step by step.
- In JavaScript, we can use functional programming techniques due to its ability to treat functions as objects.
- This allows us to pass functions as arguments, return them from other functions, and assign them to variables.

## Functional Programming in JS (II)

- JavaScript treats functions as first-class citizens, meaning they can be treated like any other data type.
  - functions can be assigned to variables
  - functions can be passed as arguments to other functions
  - functions can be returned as values from other functions
  - functions can be stored in objects or arrays
- Higher-order functions enable function composition, a key concept in functional programming.
  - Function composition is a concept in functional programming where you combine two or more functions to create a new function.

## Higher-order functions (I)

- A higher-order function is a function that can take one or more functions as arguments, return a function as its result, or both.
- Example of a higher-order function that takes a function as an argument:

```
function greet(name, country, formatterFunc) {
    return "Hello, " + name + ". Country: " + formatterFunc(country);
}

function countryAbbreviation(text) {
    return text.toUpperCase();
}

let message = greet("Ali", "ksa", countryAbbreviation);
console.log(message)
```

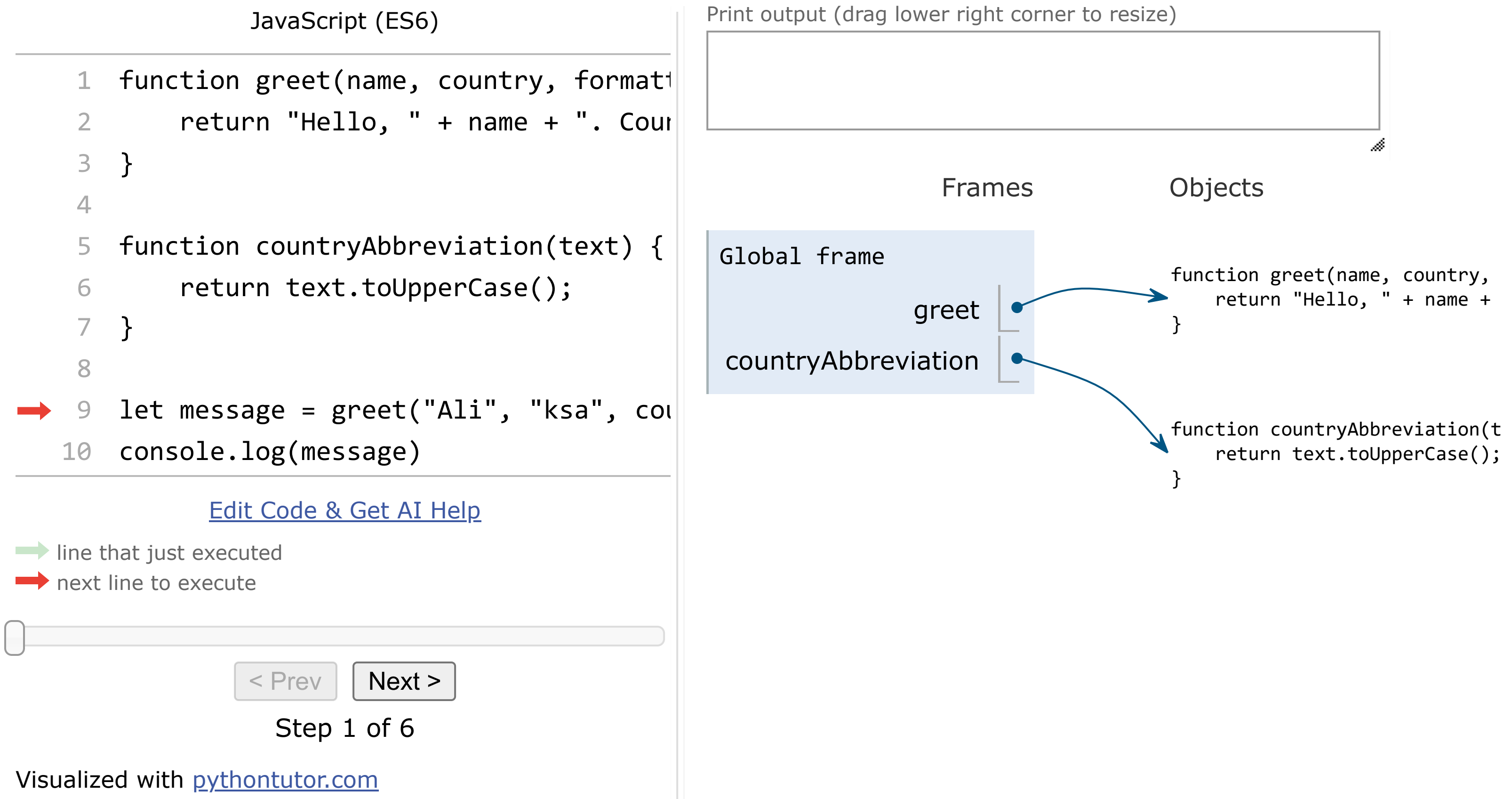
formatterFunc becomes a reference to the countryAbbreviation function, which gets called here.

greet is a higher-order function that accepts a function as its third parameter

```
Hello, Ali. Country: KSA
```

# Higher-order functions (II)

- Visualizing the execution of the previous example



JavaScript (ES6)

```
function greet(name, country, formatt
    return "Hello, " + name + ". Cour
}

function countryAbbreviation(text) {
    return text.toUpperCase();
}

let message = greet("Ali", "ksa", cou
console.log(message)
```

```
function greet(name, country,
    return "Hello, " + name +
}
```

```
function countryAbbreviation(t
    return text.toUpperCase();
}
```

Visualized with pythontutor.com

## Higher-order functions (III)

- Consider the following example of a higher-order function that returns a function as a returned value, what would the output be?

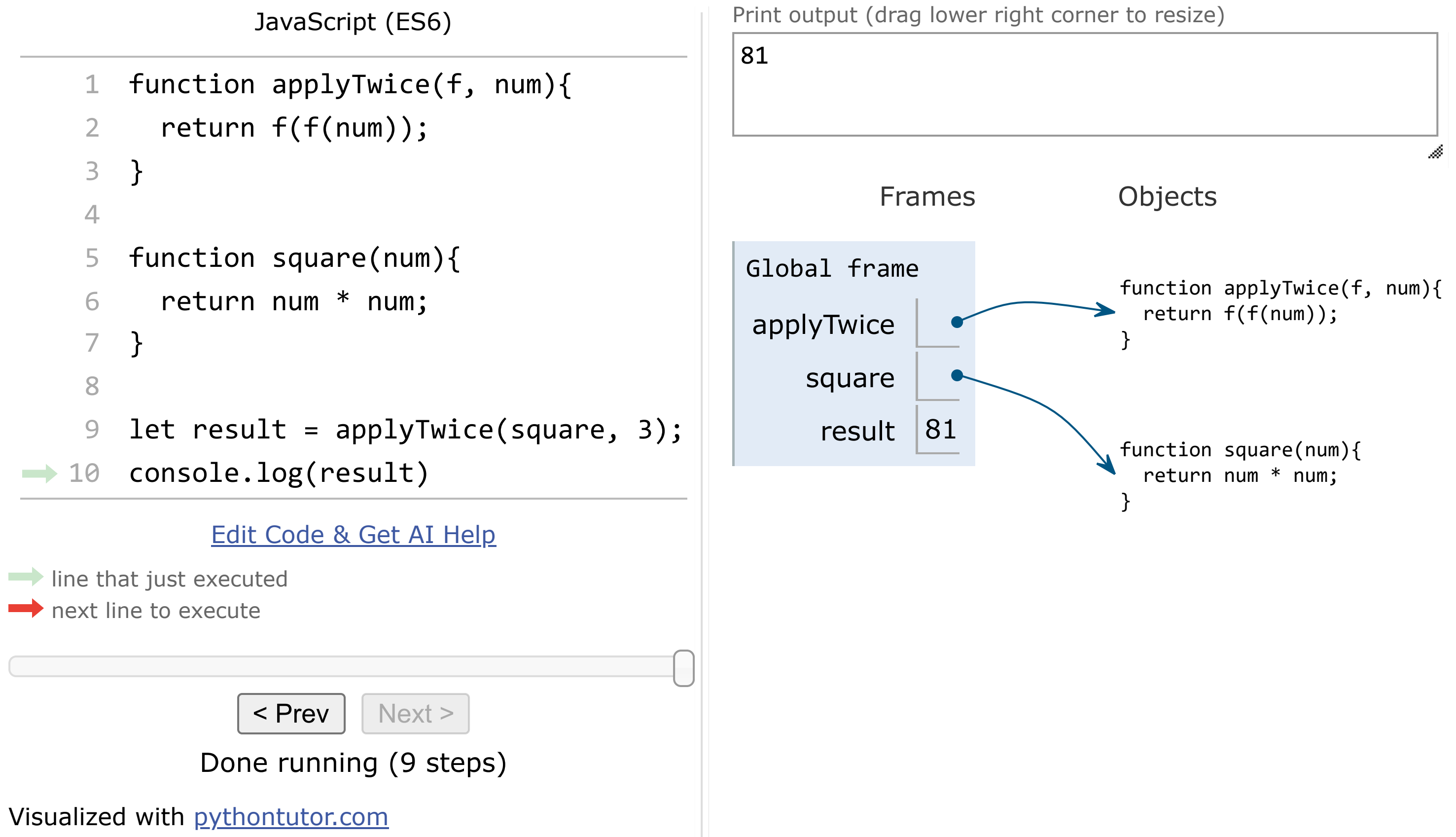
```
function applyTwice(f, num){
  return f(f(num));
}

function square(num){
  return num * num;
}

let result = applyTwice(square, 3);
console.log(result)
```

```
81
```

We'll see a graphical execution of this example in the following slide.

# Higher-order functions (III) (Cont.)

- Below is a visualized example of the previous higher-order function.



JavaScript (ES6)

```javascript
function applyTwice(f, num){
  return f(f(num));
}

function square(num){
  return num * num;
}

let result = applyTwice(square, 3);
console.log(result)
```

Visualized with pythontutor.com

Print output (drag lower right corner to resize)

```
81
```

```javascript
function applyTwice(f, num){
  return f(f(num));
}
```

```javascript
function square(num){
  return num * num;
}
```

## Higher-order functions (IV)

- Consider the following example of a higher-order function that returns a function as a returned value, what would the output be?

```
function greaterThan(n) {
  return function(numberToCheck) {
    return numberToCheck > n;
  }
}
const greaterThan10 = greaterThan(10);
let result = greaterThan10(11)
console.log(result);
result = greaterThan10(9)
console.log(result);
```

a new function is returned. This function takes one parameter and compares it with the outer function's parameter `n`.

10 is passed as the argument n to the greaterThan function

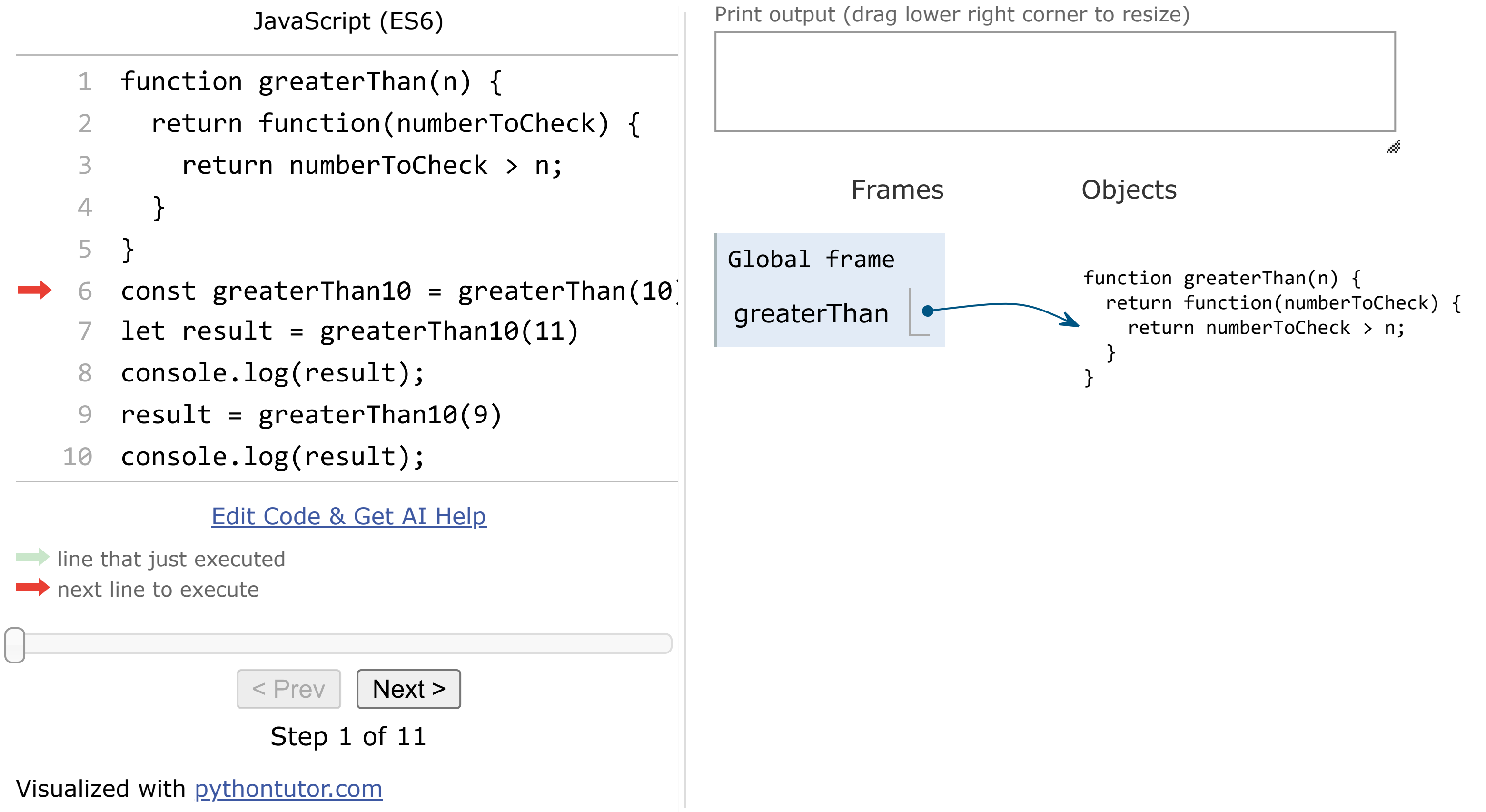
11 is passed as the argument numberToCheck to the function that was returned by greaterThan

```
true
false
```

We'll see a graphical execution of this example in the following slide.

## Higher-order functions (IV) (Cont.)

- Below is a visualized example of the previous higher-order function.



JavaScript (ES6)

```javascript
function greaterThan(n) {
  return function(numberToCheck) {
    return numberToCheck > n;
  }
}
const greaterThan10 = greaterThan(10
let result = greaterThan10(11)
console.log(result);
result = greaterThan10(9)
console.log(result);
```

Edit Code & Get AI Help

line that just executed

next line to execute

< Prev    Next >

Step 1 of 11

Visualized with pythontutor.com

Print output (drag lower right corner to resize)

Frames    Objects

Global frame

greaterThan

```javascript
function greaterThan(n) {
  return function(numberToCheck) {
    return numberToCheck > n;
  }
}
```

## Benefits of learning functional programming in JS

- **Function composition**: code is written by composing simple functions, which often results in less lines of code and more readable code.
- **Predictable code due to immutability**: Data is immutable, which means it can’t be changed after it has been created.
  - Any "changes" to the data would result in a new object being created, leaving the original data untouched.
  - This leads to fewer bugs and side effects as functions don’t alter the original data.
- **Better testing and debugging**: functional programming relies on pure functions and avoids shared data and side effects, making it easier to test and debug.
- **Framework and library support**: Many popular JavaScript libraries and frameworks, such as React and Redux, have functional programming concepts at their core.

## Array Methods for Functional Programming

| Method | Description |
|---|---|
| `map()` | Creates a new array by applying a function to every element of the original array. |
| `filter()` | Creates a new array with elements that pass a condition provided by a function. |
| `reduce()` | Applies a function to every element in of the array to reduce it to a single value. |
| `forEach()` | Executes a provided function once for each array element. |
| `some()` | Checks if at least one element in an array passes a condition implemented by the provided function. |
| `every()` | Checks if all elements in an array pass a condition implemented by the provided function. |

## `map()`

- Creates a new array by applying a function to every element of the original array.

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map(num => num * 2);
console.log(doubled);
```

```
[ 2, 4, 6, 8, 10 ]
```

## `map()` vs `for` loop

### Declarative Approach

- `map()` is often more concise and easier to read. It's often a better choice when transforming an array into another array.

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map(num => num * 2);
console.log(doubled);
```

```
[ 2, 4, 6, 8, 10 ]
```

### Imperative Approach

- A `for` loop is more verbose but it provides more control over the iteration. If you need to skip, or break out of the loop, a `for` loop is a better option.

```
const numbers = [1, 2, 3, 4, 5];
let doubled = [];
for (let i = 0; i < numbers.length; i++) {
    doubled.push(numbers[i] * 2);
}
console.log(doubled);
```

```
[ 2, 4, 6, 8, 10 ]
```

## `filter()`

- It returns a new array containing only the elements that pass the a specified condition defined in the provided function.

```
const numbers = [1, 2, 3, 4, 5];
const evenNumbers = numbers.filter(num => num % 2 === 0);
console.log(evenNumbers);
```

```
[ 2, 4 ]
```

## `reduce()`

- It reduces an array to a single value by applying a function to every element in the array.
- The `reduce` method takes two arguments: a reducer function and an initial value.

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce( (total, num) => total + num , 0);
console.log(sum);
```

```
15
```

The initial value 0 is the value from which the reduction starts. It's the value of total during the first iteration.

The reducer function `(total, num) => total + num` is an arrow function that takes two parameters: total and num. total is the accumulator that stores the ongoing total of the reduction, and num is the current array element.

## `forEach()`

- Executes a provided function once for each array element.

```
const numbers = [1, 2, 3, 4, 5];
numbers.forEach(num => console.log(num+2));
```

```
3
4
5
6
7
```

## `some()`

- Checks if at least one element in an array passes a condition implemented by the provided function.
- The provided function should return a truthy value (true or false)

```
const numbers = [1, 2, 3, 4, 5];
const hasNegativeNumbers = numbers.some(num => num < 0);
console.log(hasNegativeNumbers);
```

```
false
```

## `every()`

- Checks if all elements in an array pass a condition implemented by the provided function.
- The provided function should return a truthy value (true or false)

```
const numbers = [1, 2, 3, 4, 5];
const allPositiveNumbers = numbers.every(num => num > 0);
console.log(allPositiveNumbers)
```

```
true
```

## Wrapping up

- We have learned about the following topics:
  - **Fundamentals:** Syntax, Variables, Data Types (Strings, Numbers, Objects, Arrays), Functions, Control Flow, Loops
  - **Handling Events**: Handling Mouse & Keyboard Events
  - **DOM Manipulation:** The Document Object Model (DOM) API
  - **JSON**: converts a JavaScript object to a JSON string and vice versa.
  - **Asynchronous JavaScript:** Call back functions, Promises, Async/Await.
  - **AJAX and Fetching Data:** `XMLHttpRequest`, `fetch` API with promises and `asnc`/`await`.
  - **Functional Programming**: Array methods such as `map()`, `filter()`, `reduce()`, `forEach()`, `some()`, and `every()`.
- Coming up next: React.js ↗

## References

- Mozilla Developer Network (MDN) JavaScript. Retrieved from https://developer.mozilla.org/en-US/docs/Web/JavaScript.
- Flanagan, D. (2020). JavaScript: The Definitive Guide. 7th Edition. O’Reilly Media.
- The Modern JavaScript Tutorial. Retrieved from https://javascript.info/