

Lab 2

Lecture Review

Programming Paradigms supported in JavaScript

JavaScript is a general-purpose multi-paradigm programming language. It supports event-driven, object-oriented, and functional programming paradigms. Let's take a closer look at each.

Event-driven

Event-driven programming is a paradigm in which the flow of program execution depends upon events/messages. Instead of running your code from top to bottom and exiting, your code registers listeners for events fired by the browser. Registering listeners is usually done by providing a callback to a native function. For example:

```
btn.addEventListener("click", function () {
    console.log("Button clicked");
});

setTimeout(() => {
    alert("1 second passed");
}, 1000);
```

This paradigm is asynchronous in its nature, and care must be taken to avoid race conditions (<https://medium.com/@slavik57/async-race-conditions-in-javascript-526f6ed80665>).

Object-Oriented

Everything in JavaScript is an object, including classes! It is not like the traditional OOP (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_programming) that you know about in C++ or Java since it is not based on implementation inheritance, but rather prototypal inheritance. Instead of "copying" the implementation, each object links to the *prototype* (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes) of its parent and delegates the task (property access, method call, ...) to it if the object itself cannot perform it.

Constructors in JavaScript are just normal functions (we don't have classes that get "compiled away"). This is how to use them:

```
function Person(name) {
    this.name = name;
}

const personPrototype = {
    greet: function() {
        return `Hi! My name is ${this.name}`;
    }
};

Person.prototype = personPrototype;
Person.prototype.constructor = Person;

const john = new Person("John Smith");
console.log(john.name);
console.log(john.greet());
```

The `this` (<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this>) keyword normally refers to the function in which it is used, but it can usually get a lot more complicated than that.

The ES6 standard (released in 2015) introduced a new syntax to make the above more pleasant to write (but functionally equivalent):

```
class Person {
    constructor(name) {
        this.name = name;
    }
    greet() {
        return `Hi! My name is ${this.name}`;
    }
}

const john = new Person("John Smith");
```

Functional

Functions in JavaScript are first-class citizens. You can assign them to variables, pass them as arguments, and return them from other functions (typically referred to as a higher-order function (https://en.wikipedia.org/wiki/Higher-order_function)).

We will not go into details of the functional programming paradigm again, but here are some useful applications of this:

```
const numbers = [2, 7, 3, 9, 5, 1];
numbers.map(n => n * 2); // => [4, 14, 6, 18, 16, 2]
numbers.filter((n) => n < 5); // => [2, 3, 1]

const twice = f => x => f(f(x));
const addTwo = x => x + 2;
twice(addTwo)(5); // => 9
```

Challenge: implement map yourself

Events and the event-loop

More about events

Events (https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events) are the way your code communicates with the browser. It is a 2-way communication: the browser notifies you of events (such as button click), and you can respond to the browser with some action to do (in addition to your regular handling of the event), such as preventing the browser's default behavior or stopping its propagation/bubbling.

A common use case for preventing default behavior is to perform form validation:

```
<form id="form" action="/submit_name">
  <div>
    <label for="name">Name: </label>
    <input id="name" type="text">
  </div>
  <input id="submit" type="submit">
</form>

<script>
const form = document.getElementById('form');
const name = document.getElementById('name');
form.addEventListener('submit', function(e) {
  if (name.value === '') {
    e.preventDefault();
    alert('Name cannot be empty!');
  }
});
</script>
```

Bubbling refers to the fact that when an event occurs on a child, it **bubbles up** to the parent elements. For example:


```
<body onclick="alert('body clicked')">
  <div onclick="alert('div clicked')">
    <button onclick="alert('button clicked')">
      Click me
    </button>
  </div>
</body>
```

This will cause all 3 `onclick` handlers to run (in order from innermost to outermost) when the button is clicked, until one of them calls `event.stopPropagation()`.

Another method of controlling event dispatch is event capture. When a listener is registering to `useCapture`, it does not receive events bubbling from children, and that is should receive the events before they get dispatched to any of its children. Read more about this in the `addEventListener` (<https://developer.mozilla.org/en-US/docs/Web/API/EventTarget/addEventListener>) documentation.

The event loop

The event loop (<https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop>) is the underlying runtime model of JavaScript that actually executes the code and dispatches events. It consists of a message queue, each with an associated handler, that get handled one after the other.

 visual representation of JavaScript's runtime model

It's called a loop because its implementation usually resembles something like:

```
while (queue.waitForMessage()) {
  queue.processNextMessage()
}
```

Question: What is the output of the following?

```
console.log(1);
setTimeout(() => console.log(2), 0);
console.log(3);
```

Each message handler runs to completion before processing another message, which is why you should avoid blocking or long-running functions in your code (JavaScript is single-threaded).

You register messages in the queue when you use native functions like `addEventListener`, `setTimeout`, and so on.

Read more about the event loop and micro- and macro-tasks here (<https://javascript.info/event-loop>).

ES6 features

We already saw the class-syntax above, and saw the template literals (used for string interpolating) last lab. ES6 introduced many other useful features, such as:

Arrow functions

Arrow functions (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions) are just a much shorter way of defining functions. In short:

```
function increment(n) {  
  return n + 1;  
}  
// is equivalent to:  
const increment = n => n + 1;  
  
// if we have more than one argument, parenthesis are required  
const add = (x, y) => x + y;  
  
// for a body that's more than just one expression,  
// explicit `{}` and `return` are required  
const doSomething = () => {  
  console.log('doing something');  
  return Math.random();  
}
```

They are useful for inline functions, usually as arguments to higher-order functions. They do not affect the scope of `this`, nor do they have the `arguments` object

(<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments>).

Modules

ES6 added support for `import` and `export` in JavaScript modules

(<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules>). In short, it allows you to expose any data/functions from one file and to use them in another. Basic usage is as follows:

```
module.js
```

```
function f1() {} // not visible outside  
export function f2() {};  
export const PI = 3.14;
```

```
main.js
```

```
import { PI } from './module.js';  
console.log(PI);
```

index.html

```
<script type="module" src="main.js"></script>
```

Promises (`async / await` syntactic sugar)

A `Promise` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) is an object representing some eventual computation.

Previously, to do something with the result of a long computation, you would have to pass callbacks, leading to what's known as callback hell:

```
function makeBurger() {  
  getBeef(function(beef) {  
    cookBeef(beef, function(cookedBeef) {  
      getBuns(function(buns) {  
        putBeefBetweenBuns(buns, beef, function(burger) {  
          // Serve the burger  
        });  
      });  
    });  
  });  
};
```

Promises provide a neat interface for such asynchronous jobs that also doesn't block the main thread:

```
function makeBurger() {  
  return getBeef()  
    .then(beef => cookBeef(beef))  
    .then(cookedBeef => getBuns(cookedBeef))  
    .then(bunsWithBeef => putBeefBetweenBuns(buns, beef))  
}
```

Or even better with the `async / await` syntactic sugar (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function):

```
async function makeBurger() {  
  const beef = await getBeef();  
  const cookedBeef = await cookBeef(beef);  
  const buns = await getBuns();  
  const burger = await putBeefBetweenBuns(buns, beef);  
  return burger;  
}
```

Object destructuring

The destructuring assignment allows to extract/unpack values from objects and arrays and assign them to distinct variables.

```
const [red, green, blue] = color;  
// instead of  
const red = color[0];  
const green = color[1];  
const blue = color[2];  
  
const [num1, num2, ...rest] = [1, 2, 3, 4, 5];  
console.log(num1); // => 1  
console.log(rest); // => [3, 4, 5]  
  
function move(coords) {  
  const { x, y } = coords;  
}  
// or  
function move({ x, y }) {  
  
}  
let a, b, rest;  
({a, b, ...rest} = {a: 10, b: 20, c: 30, d: 40});  
console.log(b); // => 20  
console.log(rest); // => {c: 30, d: 40}
```

Rest/Spread operator

The `...` operator is called the spread syntax (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax), with its opposite (in meaning, but same in syntax) being called rest syntax (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters).

It allows expanding the values of an iterable in a place where multiple values are expected. For example:

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

const numbers = [1, 2, 3];
console.log(sum(...numbers)); // => 6

// (shallow-)copying an array
const numbers2 = [...numbers, 4, 5];
```

And similarly for objects, but as key-value pairs.

The rest syntax, on the other hand, collects the “remaining” parameters into an array:

```
function sum(...nums) {
  return nums.reduce((previous, current) => previous + current);
}

sum(7, 11); // => 18
sum(1, 2, 3, 4); // => 10
```

Other examples for rest syntax can be seen above in the object destructuring.

APIs

Some of the useful Web APIs (<https://developer.mozilla.org/en-US/docs/Web/API>) and built-in globals (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects) are:

- **Fetch API** (https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API): Used for performing AJAX requests
- **document** (<https://developer.mozilla.org/en-US/docs/Web/API/Window/document>): Contains useful functions for interacting with the DOM (https://developer.mozilla.org/en-US/docs/web/api/document_object_model), such as `getElementById` (<https://developer.mozilla.org/en-US/docs/Web/API/Document/getElementById>), `querySelector` (<https://developer.mozilla.org/en-US/docs/Web/API/Document/querySelector>), and `createElement` (<https://developer.mozilla.org/en-US/docs/Web/API/Document/createElement>).
- **Object** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object): Contains many utilities for interacting with objects, such as
 - `assign` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign): Copies all properties from one object to another
 - `entries` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/entries): Retrieves the key-value pairs of the object as `[key, value]` arrays

- **freeze** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze): Freezes an object such that it becomes read-only. No properties can be modified, and no extra properties can be added.
- **keys** (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys): Returns an array with the names of properties in the object

HTTP status codes

There are 5 categories of HTTP response status codes (<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>):

1. **Informational:** 100 - 199
2. **Success:** 200 - 299
3. **Redirection:** 300 - 399
4. **Client error:** 400 - 499
5. **Server error:** 500 - 599

Ones you might be familiar with (or will encounter the most) are:

- 200 (<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/200>): Ok
- 204 (<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/204>): No content
- 301 (<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/301>): Moved permanently
- 400 (<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/400>): Bad request
- 403 (<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/403>): Forbidden
- 404 (<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404>): Not found
- 418 (<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/418>): I'm a teapot
- 500 (<https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/500>): Internal server error

Interactivity with event listeners and handlers.

Let's implement a little guessing game


```
<!DOCTYPE html>
<html lang="en">
  <body>
    <form id="form">
      <label>
        Guess a number between 1 and 10 (inclusive):
        <input id="input" type="number" min="1" max="10" />
      </label>
      <button type="submit" id="guess">Guess</button>
    </form>
    <p id="result"></p>

    <script>
      let answer = Math.ceil(Math.random() * 10);

      const input = document.getElementById('input');
      const guessBtn = document.getElementById('guess');
      const form = document.getElementById('form');
      const result = document.getElementById('result');

      form.addEventListener('submit', (e) => {
        e.preventDefault();
        const guess = parseInt(input.value);
        if (Number.isNaN(guess)) {
          result.innerHTML = 'Please input a number!';
        } else if (guess === answer) {
          result.innerHTML = 'Correct answer! Thanks for playing :D';
          guessBtn.disabled = true;
          input.disabled = true;
          result.appendChild(createResetButton());
        } else if (guess < answer) {
          result.innerHTML = 'Your guess is too low :(';
        } else if (guess > answer) {
          result.innerHTML = 'Your guess is too high :/';
        }
        input.value = '';
      });

      function createResetButton() {
        const button = document.createElement('button');
        button.innerHTML = 'Play again';
        button.style.display = 'block';
        button.addEventListener('click', function resetGame() {
          answer = Math.ceil(Math.random() * 10);
          result.innerHTML = '';
          input.disabled = false;
          guessBtn.disabled = false;
        });
        return button;
      }

      /*
      input.addEventListener('keydown', (e) => {
```

```
        if (!/^\\d$/.test(e.key)) {  
            e.preventDefault();  
        }  
    });  
    */  
</script>  
</body>  
</html>
```

Note: never use `innerHTML` with unsanitized user input! It makes your code vulnerable to XSS (<https://owasp.org/www-community/attacks/xss/>).

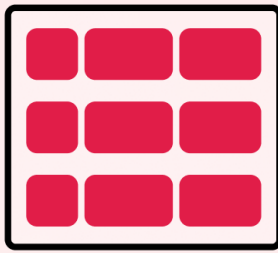
More CSS

Flexbox

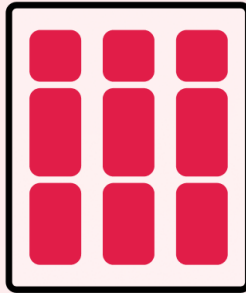
A **Flexible Box** layout arranges items in a row or column such that the items can expand to fill free space, or shrink to prevent overflow. It is 1-dimensional (unlike the grid), but it allows for wrapping.

CSS Flexbox

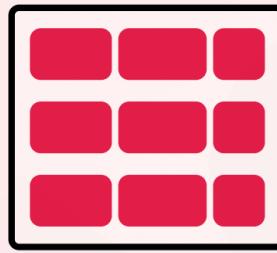
flex-direction



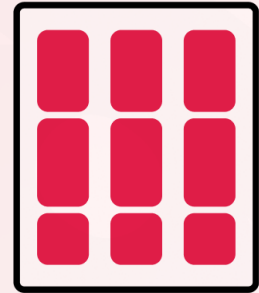
row



column

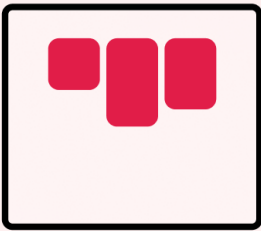


row-reverse

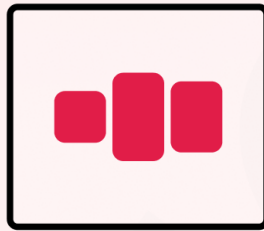


column-reverse

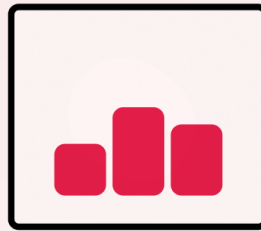
align-items



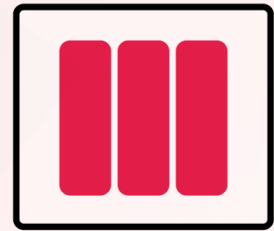
flex-start



center



flex-end



stretch

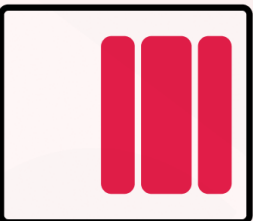
justify-content



flex-start



center



flex-end



space-between

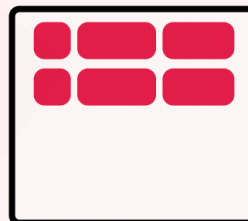


space-around

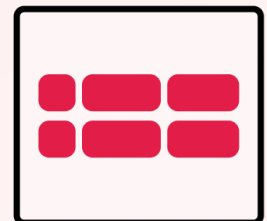


space-evenly

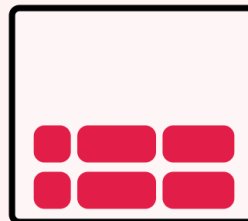
align-content



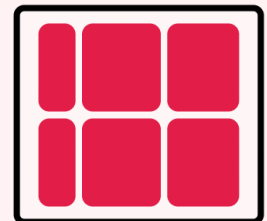
flex-start



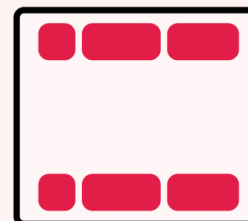
center



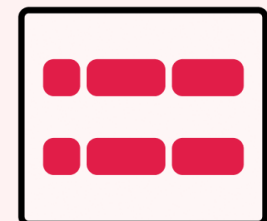
flex-end



stretch



space-between



space-around



@eludadev

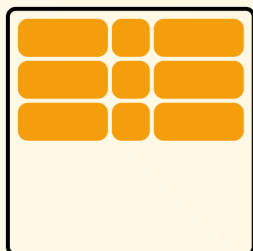
<https://css-tricks.com/snippets/css/a-guide-to-flexbox/> (<https://css-tricks.com/snippets/css/a-guide-to-flexbox/>)

Grid

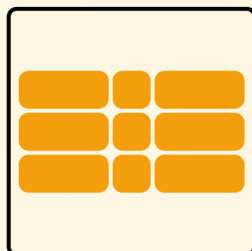
Grid is a 2-dimensional layout system consisting of rows and columns.

CSS Grid Layout

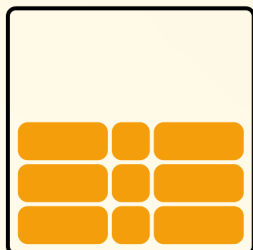
align-content



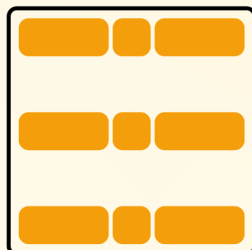
start



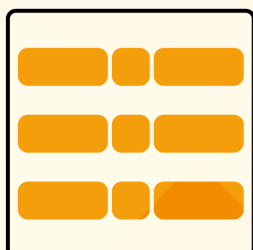
center



end



space-between

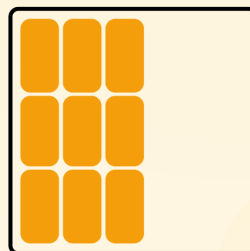


space-around

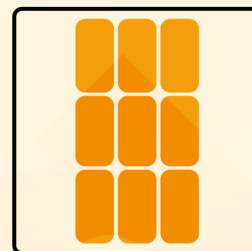


stretch

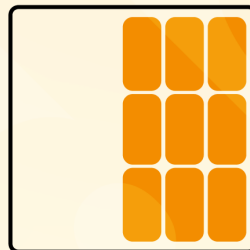
justify-content



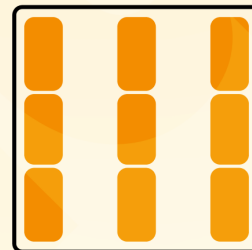
start



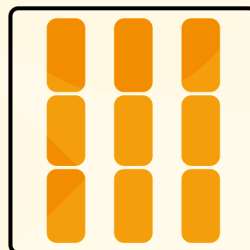
center



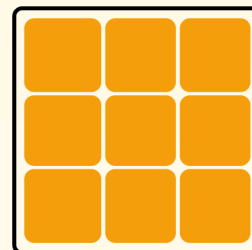
end



space-between

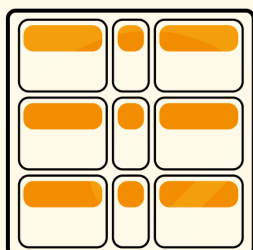


space-around

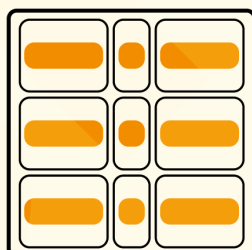


stretch

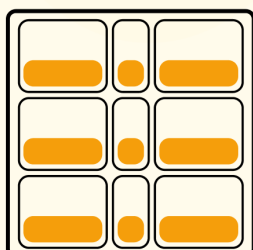
align-items



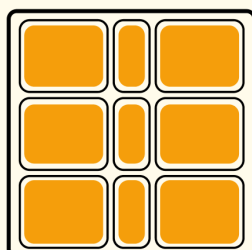
start



center

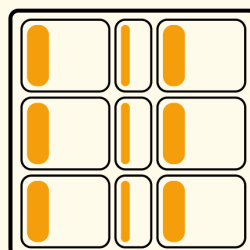


end

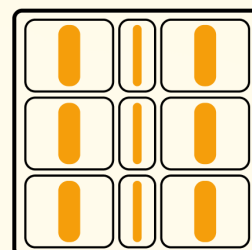


stretch

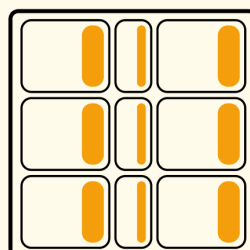
justify-items



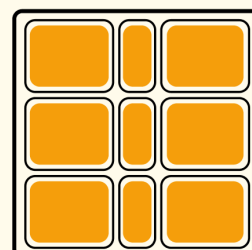
start



center



end



stretch



github.com/eludadev/css-docs

<https://css-tricks.com/snippets/css/complete-guide-grid/> (<https://css-tricks.com/snippets/css/complete-guide-grid/>)

Task

Use Fetch API to get a random joke from the JokeAPI (<https://v2.jokeapi.dev/>)


```

<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <style>
    p {
      font-size: 2em;
      padding: 1rem;
      margin: 4px;
    }
    #part2 {
      display: none;
    }
    #part2.active {
      display: block;
      color: transparent;
      background-color: lightgray;
    }
    #part2.active:hover {
      color: black;
      background-color: initial;
    }
  </style>
</head>
<body>
  <h1>Programming Jokes</h1>
  <button id="get-joke-btn">Get a random joke</button>
  <form id="type-form">
    <label><input type="radio" name="type" value="single" />Single</label>
    <br/>
    <label><input type="radio" name="type" value="twopart" />Twopart</label>
    <br />
    <label><input type="radio" name="type" value="" checked="" />Any</label>
  </form>
  <p id="part1"></p>
  <p id="part2"></p>

  <script>
    const getJokeBtn = document.getElementById('get-joke-btn');
    const part1 = document.getElementById('part1');
    const part2 = document.getElementById('part2');
    const typeForm = document.getElementById('type-form');

    function fetchSomeJoke(type) {
      const params = new URLSearchParams();
      if (type) {
        params.append('type', type);
      }
      return fetch('https://v2.jokeapi.dev/joke/Programming?' + params.toString())
        .then(r => r.json());
    }
  </script>

```

```

    }

    function handleJoke(jokeObj) {
        switch (jokeObj.type) {
            case 'single':
                return handleOnePartJoke(jokeObj);
            case 'twopart':
                return handleTwoPartJoke(jokeObj);
        }
    }

    function handleOnePartJoke(jokeObj) {
        const { joke } = jokeObj;
        part1.textContent = joke; // Note: never use innerHTML for uns
        part2.classList.remove('active');
    }

    function handleTwoPartJoke(jokeObj) {
        const { setup, delivery } = jokeObj;
        part1.textContent = setup;
        part2.classList.add('active');
        part2.textContent = delivery;
    }

    getJokeBtn.addEventListener('click', async function (e) {
        part1.textContent = part2.textContent = 'Loading...';
        const type = typeForm.elements.type.value;
        const joke = await fetchSomeJoke(type);
        handleJoke(joke);
    });
</script>
</body>
</html>

```

Homework

For the homework, we will have more practice with the Fetch API. Building on the previous homework submission, add a script that fetches an XKCD comic identifier from <https://fwd.innopolis.app/api/hw2> (<https://fwd.innopolis.app/api/hw2>) by sending your email as a query parameter (called `email`). It is recommended to use `URLSearchParams` (<https://developer.mozilla.org/en-US/docs/Web/API/URLSearchParams/URLSearchParams>).

After getting the ID, use it to request <https://xkcd.com/<your-id>/info.0.json>. You should then display the image, its title, the date it was uploaded (`date.toLocaleDateString()` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date/toLocaleDateString) is your friend). Don't forget the image's alternative text as well.

For the purpose of this assignment, let's assume that XKCD is an **untrusted** API.

Also, try to integrate flexbox and/or grid in your layout if applicable.
The exact criteria will be on Moodle.