Week 2- IntermediateJS

File and FileReader

Blob (short for Binary Large Object) is a data type that represents raw binary data or data in a format that is not necessarily text. Blobs are typically used to handle binary data, such as images, audio, video, or any other non-textual data.

A File object in JavaScript inherits from Blob and extends it with filesystem-related capabilities. There are two primary ways to obtain a File object:

1. **Using the Constructor:** You can create a File object using the constructor, which is similar to creating a Blob:

javascriptCopy code

new File(fileParts, fileName, [options]);

- **fileParts**: An array of Blob/BufferSource/String values that make up the contents of the file.
- **fileName**: A string representing the file name.
- **options** (optional): An object that can include the following property:
  - **lastModified**: An integer timestamp representing the last modification time of the file.

This constructor allows you to manually create a File object with specified content, file name, and optional last modification timestamp.

2. **From User Input:** More commonly, File objects are obtained from user interactions with the browser, such as when a user selects a file using an **<input type="file">** element or through drag-and-drop operations. In such cases, the browser automatically creates File objects with the necessary information obtained from the user's operating system. These File objects will have the following properties:
   - **name**: The file name as provided by the user.
   - **lastModified**: The timestamp representing the last modification time of the file, typically derived from the file system metadata.

When a user selects a file using an input element or through drag-and-drop, you can access these File objects directly in JavaScript.

*In summary, a File object inherits from Blob and includes additional properties like **name** and **lastModified**. You can create File objects manually using the constructor, or you can obtain them from user interactions with the browser, where the browser automatically provides the necessary file information.*

File Reader

FileReader is an object with the sole purpose of reading data from Blob (and hence File too) objects. It delivers the data using events as reading from disk may take time.

The **FileReader** object in JavaScript is used for reading the contents of files asynchronously. It provides several methods for reading files and emits events to track the progress and completion of the reading operation. Here's a short summary of the **FileReader** object and its main methods:

1. **Constructor:**
    - **let reader = new FileReader();**: Creates a new **FileReader** object.
2. **Main Methods:**
    - **readAsArrayBuffer(blob)**: Reads binary data from a Blob and returns it as an ArrayBuffer.
    - **readAsText(blob, [encoding])**: Reads data from a Blob and returns it as a text string, with an optional encoding (UTF-8 by default).
    - **readAsDataURL(blob)**: Reads binary data from a Blob and encodes it as a base64 data URL.
    - **abort()**: Cancels the ongoing reading operation.

The choice of method depends on the format of the data and how you intend to use it. Use **readAsArrayBuffer** for binary files, **readAsText** for text files, and **readAsDataURL** for embedding binary data in tags like **<img>**.

3. **Events:**
    - **loadstart**: Fired when the loading operation starts.
    - **progress**: Occurs during the reading process, providing updates on the progress.
    - **load**: Fired when the reading is successful and complete.
    - **abort**: Fired when the **abort()** method is called to cancel the operation.
    - **error**: Fired if an error occurs during the reading operation.
    - **loadend**: Fired when the reading operation is finished, whether it succeeded or failed.
4. **Result and Error:**
    - **reader.result**: Contains the result of the reading operation (if successful).
    - **reader.error**: Contains information about any error that occurred during the reading operation (if failed).

reading a file

hen working with File objects in JavaScript (which inherit from Blob), you can perform the following actions:

1. **Accessing File Properties:**
    - File objects have properties like **name** and **lastModified** in addition to the properties inherited from Blob.
    - You typically obtain File objects from user interactions, such as **<input>** elements or Drag-and-Drop events.
2. **Reading File Contents:**
    - You can use a **FileReader** object to read the contents of a File or Blob.
    - **FileReader** provides methods like **readAsText**, **readAsArrayBuffer**, and **readAsDataURL** to read the data in various formats (text, binary, or base64).
3. **Synchronous FileReader in Web Workers:**
    - In Web Workers, there is a synchronous variant called **FileReaderSync**.
    - **FileReaderSync** reads data without generating events, making it suitable for Web Workers where delays are less critical.
4. **Creating URLs:**
    - You can create URLs for File or Blob objects using **URL.createObjectURL(file)**. This allows you to display or download the file, show it as an image, or use it in various ways.
5. **Network Transfer:**
    - Sending a File over a network is straightforward using network APIs like **XMLHttpRequest** or **fetch**. These APIs natively accept File objects.

In practice, File objects are commonly used for handling user-uploaded files, reading their contents, displaying them on web pages, or sending them to servers for further processing.

Fetch

Here's a short summary of how **fetch()** works:

1. **Basic Syntax:**

- Use **fetch(url, [options])** to initiate a network request.
- **url**: The URL to access.
- **options**: Optional parameters like method, headers, etc.

2. **Promise-Based:**
   - **fetch()** returns a promise immediately, starting the request in the background.
3. **Two-Stage Process:**
   - The promise resolves with a **Response** object when the server responds with headers. You can check the HTTP status and headers at this stage.
   - The promise rejects if there are network problems or if the site doesn't exist. Even HTTP errors like 404 or 500 are considered normal flows.

Post Requests

o make a POST request or a request with another HTTP method using JavaScript's Fetch API, you need to use the **fetch** function with the following options:

1. **method**: This option specifies the HTTP method for the request, such as "POST" for creating new resources on the server. Other common methods include "GET," "PUT," "DELETE," etc.
2. **body**: The **body** option is used to provide the data you want to send with the request. It can be one of the following:
   - A string: You can send data as a JSON string or any other format you need.
   - FormData object: Used for submitting data as form/multipart, commonly used when working with HTML forms.
   - Blob/BufferSource: Used for sending binary data, such as images or files.
   - URLSearchParams: Used for submitting data in the x-www-form-urlencoded encoding, which is less common in modern web applications but still used in some scenarios.

Sending an Image

Here's a short summary of sending an image and working with Fetch options:

1. **Sending an Image**: You can submit binary data, such as an image, directly using Blob or BufferSource. For instance, you can use a **<canvas>** to draw an image and then send it to the server.
2. **Content-Type Header**: When sending binary data using a Blob object, you often don't need to manually set the **Content-Type** header because Blob objects have a built-in type (e.g., **image/png** for PNG images) that is used automatically.
3. **Async Fetch Request**: A typical Fetch request involves two **await** calls to handle the response. You can also use promise-style **.then()** syntax if you prefer.
4. **Response Properties**:
   - **response.status**: Represents the HTTP status code of the response.
   - **response.ok**: A boolean that is **true** if the status code is in the range 200-299, indicating a successful request.
   - **response.headers**: A Map-like object containing the HTTP headers of the response.
5. **Methods to Get Response Body**:
   - **response.json()**: Parses the response as a JSON object.
   - **response.text()**: Returns the response as text.
   - **response.formData()**: Returns the response as a FormData object (used for form/multipart encoding).
   - **response.blob()**: Returns the response as a Blob (binary data with a specified type).
   - **response.arrayBuffer()**: Returns the response as an ArrayBuffer (pure binary data).
6. **Fetch Options So Far**:
   - **method**: Specifies the HTTP method for the request.
   - **headers**: An object containing request headers.
   - **body**: Can be a string, FormData, BufferSource, Blob, or URLSearchParams object, depending on the data you want to send.

Day 2

What is FormData?

FormData is an essential tool in web development that enables the easy handling and transmission of form data. When you create a FormData object, it acts as a container for storing form fields and their values. This object simplifies the process of gathering data from HTML forms and sending it to a server.

Creating a FormData Object

To create a FormData object, you can use the constructor like this:

javascriptCopy code

```
let formData = new FormData([form]);
```

If you provide an HTML form element as an argument, the FormData object will automatically capture all the fields within that form. This feature makes it incredibly convenient for collecting data from user inputs.

Sending Form Data

One of the standout features of FormData is its compatibility with network methods like **fetch**. You can use a FormData object as the body of a request, and it will be encoded and sent out with the **Content-Type** header set to **form/multipart**. From the server's perspective, this appears just like a regular form submission.

FormData Methods

Summary of FormData Methods:

1. **formData.append(name, value)**: This method adds a form field with the given name and value to the FormData object. You can use it to add key-value pairs to the form data.
2. **formData.append(name, blob, fileName)**: Similar to the previous method, this one adds a field to the FormData object. However, it simulates the addition of a file input field (**<input type="file">**). The third argument, **fileName**, sets the file name as if it were the name of a file in the user's filesystem.
3. **formData.delete(name)**: This method removes a field from the FormData object with the specified name. It's useful if you need to delete a specific form field from the data.
4. **formData.get(name)**: This method retrieves the value of a field with the given name from the FormData object. You can use it to access the values of individual form fields.

5. **formData.has(name)**: This method checks whether a field with the given name exists in the FormData object. It returns **true** if a field with the specified name is present and **false** otherwise.

Additionally, there is a **formData.set(name, value)** method that has the same syntax as **append**. The difference is that **set** removes all fields with the given name and then appends a new field with the provided value. This ensures that there is only one field with that name in the FormData object, effectively overwriting any previous value or values associated with the same name.

These methods provide convenient ways to manipulate and interact with form data stored in a FormData object, making it easier to work with forms in web applications.

Sending a form with a file

Sending a Form with a File Upload

When you include an **<input type="file">** field in an HTML form and submit it, the form data is sent as **Content-Type: multipart/form-data**. This encoding is used to facilitate the uploading of files. Here's how it works:

1. **User Selects a File**: In the HTML form, when a user selects a file using the **<input type="file">** element, the browser generates a **File** object representing the selected file.
2. **Form Submission**: When the user submits the form, the browser packages the form data, including any selected files, into a **multipart/form-data** request.
3. **Server-Side Handling**: On the server side, you need to handle the **multipart/form-data** request properly to access the uploaded file(s). Popular server-side frameworks and libraries provide mechanisms for handling file uploads.
4. **File Processing**: Once you've received the **multipart/form-data** request on the server, you can access the uploaded file(s), typically through a temporary location, and process them as needed. This might involve saving the file to a specific location on the server, extracting information, or performing other operations.

Here's a simplified example of handling file uploads on the server using Node.js and the Express.js framework:

javascriptCopy code

```
const express = require("express");
const multer = require("multer"); // Middleware for handling file uploads

const app = express();
const port = 3000;

// Configure multer to handle file uploads and specify the destination folder
const storage = multer.diskStorage({
  destination: function (req, file, cb) {
    cb(null, "uploads/"); // Files will be stored in the "uploads" folder
  },
  filename: function (req, file, cb) {
    cb(null, file.originalname); // Use the original filename
  },
});

const upload = multer({ storage: storage });

// Serve HTML form with a file input
app.get("/", (req, res) => {
  res.sendFile(__dirname + "/index.html");
});

// Handle form submission with file upload
app.post("/upload", upload.single("file"), (req, res) => {
  const uploadedFile = req.file; // Access the uploaded file
  if (uploadedFile) {
    // File uploaded successfully
    res.send("File uploaded!");
  } else {
    // No file selected
    res.send("Please select a file.");
  }
});


app.listen(port, () => {
  console.log(`Server is running on port ${port}`);
});
```

The client-side HTML form might look like this:

htmlCopy code

```html
<!DOCTYPE html>
<html>
<head>
    <title>File Upload</title>
</head>
<body>
    <h1>File Upload</h1>
    <form action="/upload" method="POST" enctype="multipart/form-data">
        <input type="file" name="file">
        <input type="submit" value="Upload">
    </form>
</body>
</html>
```

When a user selects a file and submits the form, the server responds with "File uploaded!" if the file was successfully uploaded and processed.

Sending a form with Blob data

Summary:

FormData objects are a powerful tool for capturing HTML form data and sending it to a server using network methods like **fetch**.

You can create a FormData object in two ways:

1. **new FormData(form)**: This creates a FormData object by capturing all the fields within an HTML form. It's a convenient way to gather form data for submission.
2. Creating an empty object and then using methods like **formData.append(name, value)** or **formData.append(name, blob, fileName)** to add fields to it. You can also use **formData.set(name, value)** or **formData.set(name, blob, fileName)** to add fields, with the difference that the **set** method removes fields with the same name, while **append** does not.

For file uploads, you need to use the 3-argument syntax, where the last argument (**fileName**) specifies the file name, simulating the name of the file in the user's filesystem.

Other useful methods for manipulating FormData objects include:

- **formData.delete(name)**: This method removes a field with the specified name from the FormData object.
- **formData.get(name)**: It retrieves the value of a field with the given name from the FormData object.
- **formData.has(name)**: This method checks if a field with the specified name exists in the FormData object.

These methods provide flexibility in handling and modifying form data, whether you're working with simple text inputs or more complex file uploads. They make it easier to interact with forms and send data to a server for processing.

Fetch: Download progress

The **fetch** method in JavaScript allows you to track download progress when fetching data from a server. However, please note that as of the time of this explanation, there is no built-in way to track upload progress with **fetch**. For tracking upload progress, you should use XMLHttpRequest, which is covered separately.

To track download progress with **fetch**, you can utilize the **response.body** property, which provides a "readable stream" object. This stream allows you to receive the response data chunk by chunk as it arrives, giving you full control over the reading process.

Here is a step-by-step explanation of how to use **response.body** to track download progress:

1. Obtain the stream reader from **response.body** using **response.body.getReader()**.
2. Optionally, you can check the **Content-Length** header in the response to determine the total length of the response, which can help you calculate progress.
3. Use a loop with **await reader.read()** to read chunks of data until **done** is **true**. The result of this call is an object with two properties: **done** (a boolean indicating if reading is complete) and **value** (a **Uint8Array** containing the bytes of the chunk).
4. Accumulate the response chunks in an array. This is important because once the response is consumed, you cannot re-read it.
5. After reading all the chunks, you need to concatenate them into a single result. This involves creating a new **Uint8Array** with the combined length and copying each chunk into it using **.set()**.
6. If you are working with JSON data, you can use **TextDecoder** to interpret the bytes and then parse the resulting string with **JSON.parse()**.
7. If you need binary content instead of JSON, you can create a blob from all the chunks using **new Blob(chunks)**.

This approach allows you to not only fetch data but also track the download progress in the process. However, please be aware that this method is specifically for download progress and does not apply to tracking upload progress when sending data to a server.

Fetch: Abort

Summary of Aborting Fetch with AbortController:

Aborting a fetch request in JavaScript can be achieved using the **AbortController** object, as fetch returns a promise that does not inherently support cancellation. Here's how to use **AbortController** to cancel a fetch:

Step 1: Create a Controller

let controller = new AbortController();

Day 3

Patterns and flags

A regular expression, often abbreviated as "regexp" or "reg," consists of a pattern and optional flags in JavaScript. There are two ways to create a regular expression object: the long syntax using the **new RegExp("pattern", "flags")** format and the short syntax using slashes **//**, which are used to enclose the pattern and optionally include flags. The slashes indicate to JavaScript that a regular expression is being created, much like how quotes are used for strings. Regular expressions are powerful tools for pattern matching and manipulation in JavaScript.

Usage

In JavaScript, you can use regular expressions to search for patterns within strings. You can create regular expressions using either the short syntax **//** or the long syntax **new RegExp("pattern", "flags")**.

In the short syntax, you enclose the pattern in slashes, like **/love/**, and it's equivalent to performing a simple substring search for "love" within the string.

let str = "I love JavaScript!";

let regexp = /love/;

alert(str.search(regexp)); // 2

The **str.search(regexp)** method returns the position of the first match of the regular expression within the string.

You would use the long syntax, **new RegExp("pattern", "flags")**, when you need to create a regular expression dynamically from a string, which is useful for cases where the pattern is not known beforehand.

let tag = prompt("Which tag you want to search?", "h2");

let regexp = new RegExp(tag);

// Dynamically searches for the specified tag, e.g., <h2>

alert("<h1> <h2> <h3>".search(regexp));

The short syntax is commonly used for simple, static patterns, while the long syntax offers more flexibility when patterns need to be generated dynamically based on user input or other variables.

Flags

In JavaScript, regular expressions can have flags that affect how they perform searches. There are six primary flags in JavaScript:

1. **i (Case-Insensitive):**

- With the **i** flag, the search is case-insensitive, meaning there is no distinction between uppercase and lowercase characters when matching.
- Example: **/LOVE/i** will match "love" or "LOVE" in the input string.

2. **g (Global Search):**
   - The **g** flag tells the regular expression to search for all matches in the input string rather than stopping after the first match.
   - Example: **/pattern/g** will find all occurrences of "pattern" in the input string.
3. **m (Multiline Mode):**
   - The **m** flag, when used, affects how the **^** and **$** anchors work. It allows them to match the start and end of each line within a multi-line input string.
   - This flag is covered in more detail in the context of multiline matching.
4. **s (Dotall Mode):**
   - The **s** flag, also known as "dotall" mode, makes the dot (**.**) in the regular expression match any character, including newline characters (**\n**).
   - This flag is covered in the context of character classes.
5. **u (Unicode Support):**
   - The **u** flag enables full Unicode support in regular expressions. It ensures proper handling of Unicode characters, including surrogate pairs.
   - Useful when dealing with non-ASCII characters and complex Unicode text.
6. **y (Sticky Mode):**
   - The **y** flag, known as "sticky" mode, restricts the regular expression search to start only at the specified position in the input string, as opposed to searching the entire string.
   - This flag is covered in more detail in the context of sticky searching.

Methods of RegExp and String

In JavaScript, there are various methods provided by the **RegExp** class and regular strings to work with regular expressions. Here's a summary of these methods and their common use cases:

To search for all matches:

- Use the **g** flag with the regular expression.
- Get a flat array of matches: **str.match(reg)**.
- Get an array of matches with details: **str.matchAll(reg)**.

To search for the first match only:

- Use the regular expression without the **g** flag.
- Get the full first match: **str.match(reg)**.
- Get the string position of the first match: **str.search(reg)**.
- Check if there's a match: **regexp.test(str)**.
- Find the match from a given position: Set **regexp.lastIndex** to the desired position and use **regexp.exec(str)**.

To replace all matches:

- Replace with another string or a function result: **str.replace(reg, str|func)**.

To split the string by a separator:

- Use **str.split(str|reg)**.

Additional methods and details:

- The **str.match(reg)** method behaves differently depending on the presence of the **g** flag in the regular expression.
- The **str.matchAll(reg)** method returns an iterable, and you can use **Array.from(result)** or a **for..of** loop to work with it.
- The **str.split(regexp|substr, limit)** method splits the string using the regular expression as a delimiter.
- The **str.replace(str|reg, str|func)** method is a versatile tool for searching and replacing, and you can use special placeholders like **$&**, **$1**, **$2**, etc., in the replacement string.
- You can use a function as the replacement argument in **str.replace()** for more complex replacements.
- The **regexp.exec(str)** method is a flexible searching method that returns the first match and allows you to search from a given position when using the **g** flag.
- The **regexp.test(str)** method checks if a match exists and returns **true** or **false**.

Remember that the behavior of some methods depends on the presence of the **g** flag in the regular expression. Understanding these methods and their usage patterns is essential for effective text manipulation and searching using regular expressions in JavaScript.

Character classes

Character classes in regular expressions allow you to match specific types of characters within a pattern. Here are the most commonly used character classes in JavaScript:

1. **\d:** Matches any single digit (0-9).
2. **\s:** Matches any whitespace character, including spaces, tabs, and newlines.

3. **\w:** Matches a "wordly" character, which includes letters of the English alphabet, digits, and underscores. Non-Latin characters (e.g., Cyrillic or Hindi) do not belong to \w.

Character classes can be combined with other regular symbols in a regular expression pattern to match specific sequences of characters. For example, **\d\s\w\w\w\w\d** matches a digit followed by a space character followed by four wordly characters and another digit.

These character classes are useful for tasks like extracting specific types of information from strings, as demonstrated in the example provided.

Word boundary \b

A word boundary **\b** is a special character class in regular expressions that doesn't represent a character but rather a boundary between characters. It is used to ensure that a pattern is matched at the boundary of words in a string. Here are the key characteristics of word boundaries:

- **\b** checks for a word boundary, which can be one of three variants:
    1. Immediately before is a word character (**\w**), and immediately after is not a word character (or vice versa).
    2. At the string's start, and the first character is a word character.
    3. At the string's end, and the last character is a word character.
- Word boundaries are used to match complete words and are commonly employed when searching for standalone English words in text.

For example, **\bJava\b** will match "Java" in the string "Hello, Java!" but not in the string "Hello, JavaScript!" because it ensures that "Java" is a standalone word.

Word boundaries are helpful for tasks such as finding standalone words, detecting two-digit numbers, and more in text. However, they are specific to word characters in the English alphabet and may not work as expected for non-Latin alphabets. Unicode character classes can be used for broader language support.

Inverse Classes

Inverse classes in regular expressions are character classes that match characters not belonging to a specific category. They are denoted by the same letter as the original class but in uppercase. Here are some common inverse character classes:

1. **\D:** Matches any character that is not a digit. It is the inverse of \d.
2. **\S:** Matches any character that is not a whitespace character (e.g., space, tab, newline). It is the inverse of \s.
3. **\W:** Matches any character that is not a "wordly" character, which includes letters, digits, and underscores (\w). It matches any non-word character.

4. **\B:** Matches a position that is not a word boundary (\b).

Inverse classes are useful when you want to find or replace characters that do not belong to a specific category. For example, using **\D** can help you extract all non-digit characters from a string, and using **\S** can help you find all non-whitespace characters.

Spaces are regular characters

Spaces are regular characters in a regular expression. They are treated equally with any other character in the pattern. If spaces are not explicitly included in the regular expression pattern, they will not be matched. This means that spaces must be considered and included in the regular expression pattern when searching for specific patterns in a string. Extra spaces or any other characters that are not accounted for in the regular expression may prevent a match. So, in a regular expression, all characters, including spaces, are significant and need to be accounted for to ensure accurate matching.

A dot is any character

The dot (".") in a regular expression is a special character class that matches "any character except a newline." It matches any character, including letters, digits, symbols, and spaces. However, it does not match the absence of a character; there must be a character in the string for the dot to find a match. Here's a summary:

- The dot (".") in a regular expression matches any character except a newline character.
- It can be used within a regular expression pattern to match any single character.
- The dot does not match the absence of a character; there must be a character in the string for it to find a match.

The dotall "s" flag

The dotall "s" flag is a regular expression flag that changes the behavior of the dot (".") metacharacter. By default, the dot matches any character except a newline character ("\n"). However, when the "s" flag is used in a regular expression, the dot matches literally any character, including newline characters.

Here are some key points about the dotall "s" flag and other character classes:

- By default, the dot does not match newline characters ("\n").
- The "s" flag changes the dot's behavior to match any character, including newline characters ("\n").
- Character classes like \d (digits), \D (non-digits), \s (space symbols including tabs and newlines), \S (all characters except space symbols), \w (English letters, digits, underscore '_'), and \W (all characters except \w) are useful for various character matching tasks.
- In addition to the default character classes, modern JavaScript supports Unicode properties for characters. For example, you can use \p{Script=Cyrillic} or

\p{sc=Cyrillic} to match Cyrillic letters, \p{Dash_Punctuation} or \p{pd} to match dashes, or \p{Currency_Symbol} or \p{sc} to match currency symbols. Unicode provides a wide range of character categories that can be selected for matching specific characters.

Using the "s" flag and Unicode properties, you can create powerful regular expressions that match characters based on their properties, making it easier to work with text in various languages and scripts.

Escaping, Special characters

1. Special Characters: In regex, certain characters have special meanings, such as **[\ ^ $. | ? * + ( )]**.
2. Escaping: To use a special character as a literal character in a regex, you should prepend it with a backslash (). This is called "escaping."
3. Examples:
   - To match a literal dot (.), use **\.**.
   - To match parentheses, use **\(** and **\)**.
   - To match a backslash (), use \\.
   - To match a slash (/), you should escape it within **/.../**, but not when creating a regex with **new RegExp**.
4. Slash in Regex:
   - Inside **/.../**, you need to escape the slash, like **/\//**.
   - When using **new RegExp**, you don't need to escape the slash, e.g., **new RegExp("/")**.
5. Escaping in **new RegExp**:
   - When creating a regex using **new RegExp**, be aware that backslashes in strings are interpreted by the string itself.
   - To fix this, double backslashes in the string before passing it to **new RegExp**.

In summary, to search for special characters as literal characters in regex, use backslashes to escape them. Be mindful of how backslashes are treated in regular strings and when creating regex objects with **new RegExp**.