MODERN JAVASCRIPT

ADVANCED INTERNET APPLICATIONS

ASYNCHRONOUS JAVASCRIPT

JAVASCRIPT EXECUTION

- JavaScript is single-threaded
- Long operations block the main thread
- Need for asynchronous programming patterns

CALLBACKS - TRADITIONAL APPROACH

```
function getData(url, successCallback,
errorCallback) {
  const xhr = new XMLHttpRequest();
  xhr.open('GET', url);

  xhr.onload = function() {
    if (xhr.status === 200) {
      successCallback(xhr.responseText);
    } else {
      errorCallback('Request failed: '
        + xhr.status);
    }
  };

  xhr.onerror = function() {
    errorCallback('Network error');
  };

  xhr.send();
}
```

```
getData(
  'https://api.sampleapis.com/coffee/hot',
  function(data) {
    console.log('Success:', data);
  },
  function(error) {
    console.error('Error:', error);
  }
);
```

CALLBACK HELL

```
getData('users.json', function(usersData) {
  getPermissions(usersData.adminId, function(permissionsData) {
    getActivities(permissionsData.activityId, function(activitiesData) {
      getLog(activitiesData.logId, function(logData) {
        // Deeply nested, hard to read and maintain
        console.log(logData);
      }, handleError);
    }, handleError);
  }, handleError);
}, handleError);
```

Problems:

- Hard to read (pyramid of doom)
- Error handling is repetitive
- Flow control is difficult

PROMISES

A Promise represents an operation that hasn't completed yet, but is expected to in the future.

```
const promise = new Promise((resolve, reject) => {
  // Asynchronous operation here
  if (/* operation successful */) {
    resolve(value); // Success
  } else {
    reject(error);  // Failure
  }
});
```

Three states:

- Pending - Initial state, neither fulfilled nor rejected
- Fulfilled - Operation completed successfully
- Rejected - Operation failed

USING PROMISES

```
function getData(url) {
  return new Promise((resolve, reject) => {
    const xhr = new XMLHttpRequest();
    xhr.open('GET', url);

    xhr.onload = function() {
      if (xhr.status === 200) {
        resolve(xhr.responseText);
      } else {
        reject('Request failed: '
          + xhr.status);
      }
    };

    xhr.onerror = function() {
      reject('Network error');
    };

    xhr.send();
  });
}
```

```
getData('https://api.sampleapis.com/coffee/hot'
  .then(data => console.log('Success:', data))
  .catch(err => console.error('Error:', err));
```

PROMISE CHAINING

```
// Instead of callback hell
getData('users.json')
  .then(userData => {
    console.log('Got user data');
    return getPermissions(userData.adminId);
  })
  .then(permissionsData => {
    console.log('Got permissions');
    return getActivities(permissionsData.activityId);
  })
  .then(activitiesData => {
    console.log('Got activities');
    return getLog(activitiesData.logId);
  })
  .then(logData => {
    console.log('Final result:', logData);
  })
  .catch(error => {
    console.error('Error in chain:', error);
  });
```

PROMISE METHODS

```
// Promise.all - Wait for all promises to resolve
Promise.all([
  fetch('users.json'),
  fetch('products.json'),
  fetch('orders.json')
])
.then(responses => Promise.all(responses.map(r => r.json())))
.then(([users, products, orders]) => {
  console.log('All data loaded:', users, products, orders);
})
.catch(error => console.error('Error loading data:', error));

// Promise.race - Wait for the first promise to resolve or reject
Promise.race([
  fetch('fast-server.com/data'),
  fetch('slow-server.com/data')
])
.then(response => console.log('Got first response!'))
.catch(error => console.error('Error:', error));
```

ASYNC/AWAIT

Syntactic sugar over Promises that makes async code look synchronous

```
// Async function always returns a Promise
async function fetchData() {
  try {
    // Await pauses execution until promise resolves
    const user = await fetch('users.json').then(r => r.json());

    const perm = await fetch(`perm/${user.adminId}.json`).then(r => r.json());

    const act = await fetch(`act/${perm.activityId}.json`).then(r => r.json());

    const log = await fetch(`logs/${act.logId}.json`).then(r => r.json());

    return log;
  } catch (error) {
    console.error('Error:', error);
    throw error; // Re-throw to allow further catch
  }
}

fetchData()
  .then(res => console.log('Result:', res))
  .catch(err => console.error('Caught:', err));
```

ERROR HANDLING WITH ASYNC/AWAIT

```
async function fetchWithErrorHandling() {
  try {
    const response = await fetch('https://api.sampleapis.com/coffee/hot');
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }

    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Fetching data failed:', error);

    if (error.name === 'TypeError') {
      console.log('Network issue - check your connection');
    }

    // Re-throw or return a default value
    throw error;
  } finally {
    console.log('Fetch operation completed');
  }
}
```

THE FETCH API

Modern replacement for XMLHttpRequest

```
fetch('https://api.sampleapis.com/coffee/hot')
  .then(response => {
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }
    return response.json(); // Parse JSON response
  })
  .then(data => {
    console.log('Data:', data);
  })
  .catch(error => {
    console.error('Fetch error:', error);
  });
```

FETCH OPTIONS

```
// POST request with JSON body
fetch('https://api.example.com/users', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer token123'
  },
  body: JSON.stringify({
    name: 'John Doe',
    email: 'john@example.com'
  })
})
.then(response => response.json())
.then(data => console.log('Created user:', data))
.catch(error => console.error('Error:', error));
```

FETCH WITH ASYNC/AWAIT

```
async function postData(url, data) {
  try {
    const response = await fetch(url, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify(data)
    });

    if (!response.ok) {
      throw new Error(`HTTP error!
      Status: ${response.status}`);
    }

    return await response.json();
  } catch (error) {
    console.error('Error:', error);
    throw error;
  }
}
```

```
async function createUser() {
  try {
    const newUser = await postData(
        'https://api.example.com/users', {
      name: 'Jane Smith',
      email: 'jane@example.com'
    });
    console.log('User created:', newUser);
  } catch (error) {
    console.error('Failed user:', error);
  }
}
```

JAVASCRIPT MODULES

WHAT ARE MODULES?

- Self-contained pieces of code
- Explicit dependencies
- Explicit exports
- Encapsulated scope
- Reusable across files

MODULE SYNTAX - EXPORT

```
// Named exports
export const PI = 3.14159;
export function square(x) {
  return x * x;
}

// Or grouped exports
const e = 2.71828;
function cube(x) {
  return x * x * x;
}
export { e, cube };

// Default export (only one per module)
export default function calculate(operation, a, b) {
  // Implementation
}

// Renaming exports
function add(a, b) { return a + b; }
export { add as sum };
```

MODULE SYNTAX - IMPORT

```
// Import named exports
import { PI, square } from './math.js';
console.log(PI, square(4)); // 3.14159, 16

// Import default export
import calculate from './calculator.js';
calculate('add', 2, 3); // 5

// Import both default and named exports
import calculate, { PI, square } from './all-math.js';

// Rename imports
import { square as getSquare } from './math.js';
console.log(getSquare(4)); // 16

// Import all exports as a namespace object
import * as math from './math.js';
console.log(math.PI, math.square(4)); // 3.14159, 16
```

MODULE BENEFITS

- Avoid global namespace pollution
- Better code organization
- Explicit dependencies
- Reusable code
- Easier maintenance
- Better for larger applications

USING MODULES IN BROWSERS

```
<!-- Add type="module" to script tag -->
<script type="module">
  import { formatDate } from './utils.js';
  console.log(formatDate(new Date()));
</script>

<!-- Or link to a module script -->
<script type="module" src="app.js"></script>
```

Notes:

- Modules are automatically in strict mode
- Modules have their own scope (no global variables)
- Modules are deferred by default
- Cannot use modules directly from the filesystem (CORS restriction) - need a local server

ORGANIZING CODE WITH MODULES

```
project/
  ├── index.html
  ├── app.js              // Main entry point
  ├── services/
  │   ├── api.js          // API interactions
  │   └── auth.js         // Authentication
  ├── components/
  │   ├── user-card.js    // UI components
  │   └── navigation.js
  └── utils/
      ├── date-format.js  // Utility functions
      └── validation.js
```

Example import paths:

```
// In app.js
import { fetchUsers } from './services/api.js';
import { UserCard } from './components/user-card.js';
```

THANK YOU!

QUESTIONS?