CSS233 WEB PROGRAMMING I

LECTURE 8 Asynchronous Programming

Dr. Wittawin Susutti
wittawin.sus@kmutt.ac.th

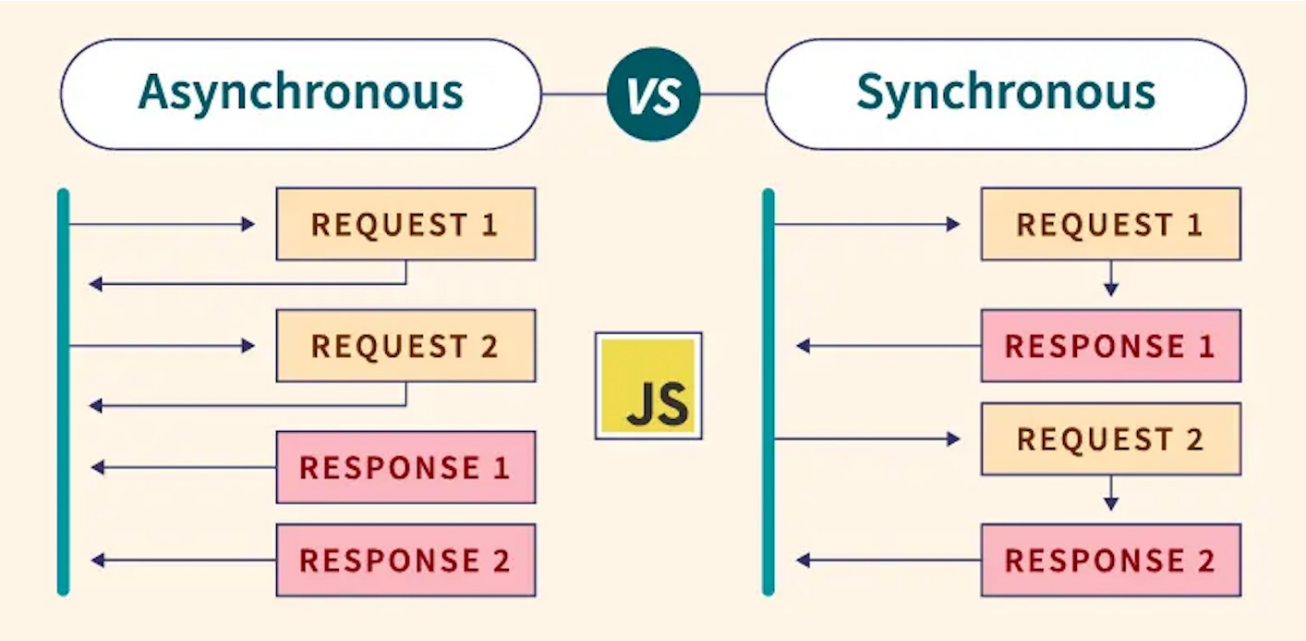
Synchronous Programming

- In a synchronous system, tasks are completed one after another.
- Think of this as if you have just one hand to accomplish 10 tasks. So, you have to complete one task at a time.
- JavaScript is by default Synchronous **[single threaded]**. Think about it like this – one thread means one hand with which to do stuff.

Asynchronous Programming

- In modern systems, lots going on at once
  - Computation
  - Animation
  - User input
  - Network communication
- Often need to start something now and finish it later
- Imagine that for 10 tasks, you have 10 hands. So, each hand can do each task independently and at the same time.

Asynchronous VS Synchronous



Two-edged Sword

Benefits

- Exploit available parallelism
- Do 10 things are once, finish 10 times faster
- Continue to be responsive while waiting for something slow

Costs

- Want result immediately, have to wait
  - stay responsive to user while waiting
  - have to remember what I was waiting for
- Asynchrony is confusing
  - code is complex, hard to read
  - source of many bugs
  - hard to debug
  - e.g. tracing execution is naturally sequential

Example: Script Loading

- Standard script loading is **synchronous**
  - page load pauses while script is fetched, executed
  - one thing at a time
- Script at beginning loads/runs **before** page complete
  - delays parsing of rest of page
  - cannot count on page being present
  - must take care manipulating page
- Script at end loads/runs **after** page complete
  - Delays download of script
  - Delays execution of script
  - May reduce responsiveness

Example: Script Loading

```
<script src="early.js">
<div>
       content here...
</div>
<script src="late.js">
```

```
<script src="s1.js"></script>
<!--page fetch pauses here while s1.js is fetched
and executed-->
<script defer src="s3.js"></script>
<!--page fetch continues while s3.js is fetched-->
<!--s3.js executes after page is loaded-->
<!--fetch in parallel, execute in sequence-->
<script type="module" src="s3.js"></script>
<!--implicitly deferred-->
<script async src="s2.js"></script>
<!--page fetch/exec continues while s2.js is
fetched and executed-->
<!--fetch/execute both in parallel—more parallel
than defer-->
```

More General Solutions

- **`async`** and **`defer`** are special syntax for common problem of script loading
- What about general asynchronous computing?
  - Objects/methods for asynchrony
    - Callbacks
    - Promises
  - Syntax for asynchrony
    - `async/await`: intuitive syntax

Callbacks

- Want to do something when X finishes
- Arrange for X to tell you when it's done
- By invoking a **callback function** you give to X
  - By passing it as an argument to X
- When callback is invoked, result of X can be passed as an argument
- When you nest a function inside another function as an argument, that's called a callback.

Callbacks

- A function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.
  - Synchronous
  - Asynchronous

```
function logQuote(quote) {
       console.log(quote);
}

function createQuote(quote, callback) {
       const myQuote = `Like I always say,
                        '${quote}'`;
       callback(myQuote);
}

createQuote("WebApp I rocks!", logQuote);
```

Asynchronicity

- JavaScript is single-threaded and inherently synchronous
  - i.e., code cannot create threads and run in parallel in the JS engine
- Callbacks are the most fundamental way for writing asynchronous JS code
- How can they work asynchronously?
  - e.g., how can `setTimeout()` or other async callbacks work?
- Thanks to the Execution Environment
  - e.g., browsers and Node.js
- and the Event Loop

```
const deleteAfterTimeout = (task) => {
       // do something
}
// runs after 2 seconds
setTimeout(deleteAfterTimeout, 2000, task)
```

Non-Blocking Code

- Asynchronous techniques are very useful, particularly for web development
- For instance: when a web app runs executes an intensive chunk of code without returning control to the browser, the browser can appear to be frozen
  - this is called blocking, and it should be the exception!
    - the browser is blocked from continuing to handle user input and perform other tasks until the web app returns control of the processor
- This may happen outside browsers, as well
  - e.g., reading a long file from the disk/network, accessing a database and returning data, accessing a video stream from a webcam, etc.
- Most of the JS execution environments are, therefore, deeply asynchronous
  - with non-blocking primitives
  - JavaScript programs are event-driven, typically

Asynchronous Callbacks

- The most fundamental way for writing asynchronous JS code
- Handling user actions
- Handling I/O operations
- Handling time intervals
- Interfacing with databases

```
const readline = require('readline’);

const rl = readline.createInterface({
       input: process.stdin,
       output: process.stdout
});

rl.question('How old are you? ', (answer) => {
       let description = answer;
       rl.close();
});
```

Timers

- To delay the execution of a function:
  - `setTimeout()` runs the callback function after a given period of time
  - `setInterval()` runs the callback function periodically

```
const onesec = setTimeout(()=> {
       console.log('hey') ; // after 1s
}, 1000) ;

console.log('hi') ;
```

```
const myFunction = (firstParam, secondParam) => {
       // do something
}
// runs after 2 seconds
setTimeout(myFunction, 2000, firstParam, secondParam) ;
```

Timers

- `clearInterval()`: for stopping the periodical invocation of setInterval

```
const id = setInterval(() => {}, 2000);
// «id» is a handle that refers to the timer

clearInterval(id) ;
```

Handling Errors in Callbacks

- No “official” ways, only best practices.
- Typically, the first parameter of the callback function is for storing any error, while the second one is for the result of the operation
  - this is the strategy adopted by Node.js, for instance

```
fs.readFile('/file.json', (err, data) => {
      if (err !== null) {
            console.log(err);
            return;
      }
      //no errors, process data
      console.log(data);
});
```

Callback Hell

- If you want to perform multiple asynchronous actions in a row using callbacks, you must keep passing new functions to handle the continuation of the computation after the previous action
  - every callback adds a level of nesting
  - when you have lots of callbacks, the code starts to be complicated very quickly

```
import readline from 'readline’;
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question('Task description: ', (answer) => {
  let description = answer;
  rl.question('Is the task important? (y/n)', (answer) => {
    let important = answer;
    rl.question('Is the task private? (y/n)', (answer) => {
      let privateFlag = answer;
        rl.question('Task deadline: ', (answer) => {
          let date = answer;
          ...
          rl.close();
        })
    })
  })
});
```

Promises

- A core language feature to “simplify” asynchronous programming
  - a possible solution to callback hell, too!
  - a fundamental building block for “newer” functions (async, ES2017)
- It is an object representing the eventual completion (or failure) of an asynchronous operation
  - i.e., an asynchronous function returns a promise to supply the value at some point in the future, instead of returning immediately a final value
- Promises standardize a way to handle errors and provide a way for errors to propagate correctly through a chain of promises

Promises

- Promises can be created or consumed
  - many Web APIs expose Promises to be consumed!
- When consumed:
  - a Promise starts in a pending state
    - the caller function continues the execution, while it waits for the Promise to do its own processing, and give the caller function some “responses”
  - then, the caller function waits for it to either return the promise in a fulfilled state or in a rejected state

Creating a Promise

- A Promise object is created using the `new` keyword
- Its constructor takes an executor function, as its parameter
- This function takes two functions as parameters:
  - `resolve`, called when the asynchronous task completes successfully and returns the results of the task as a value
  - `reject`, called when the task fails and returns the reason for failure (an error object, typically)

```
const myPromise =
  new Promise((resolve, reject) => {

    // do something asynchronous which
    // eventually call either:

      resolve(someValue); // fulfilled

    // or

      reject("failure reason"); // rejected
});
```

Creating a Promise

- You can also provide a function with “promise functionality”
- Simply have it return a promise

```
function waitPromise(duration) {
  // Create and return a new promise
  return new Promise((resolve, reject) => {
      // If the argument is invalid,
      // reject the promise
    if (duration < 0) {
      reject(new Error('Time travel not yet implemented'));
    } else {
      // otherwise, wait asynchronously and then resolve the
      // Promise; setTimeout will invoke resolve() with no
      // arguments: the Promise will fulfill with the
      // undefined value
      setTimeout(resolve, duration);
    }
  });
}
```

Consuming a Promise

- When a Promise is fulfilled, the `then()` callback is used
- If a Promise is rejected, instead, the `catch()` callback will handle the error
- `then()` and `catch()` are instance methods defined by the Promise object
  - each function registered with `then()` is invoked only once
- You can omit `catch()`, if you are interested in the result, only

```
waitPromise().then((result) => {
 console.log("Success: ", result);
}).catch((error) => {
 console.log("Error: ", error);
});

// if a function returns a Promise...
waitPromise(1000).then(() => {
 console.log("Success!");
}).catch((error) => {
 console.log("Error: ", error);
});
```

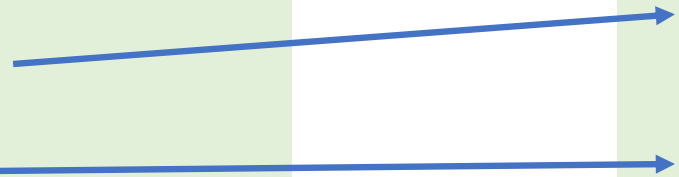
Consuming a Promise

- `p.then(onFulfilled[, onRejected]);`
  - Callbacks are executed asynchronously (inserted in the event loop) when the promise is either fulfilled (success) or rejected (optional)
- `p.catch(onRejected);`
  - Callback is executed asynchronously (inserted in the event loop) when the promise is rejected
  - Short for `p.then(null, failureCallback)`
- `p.finally(onFinally);`
  - Callback is executed in any case, when the promise is either fulfilled or rejected
  - Useful to avoid code duplication in then and catch handlers
- All these methods return Promises, too! ⇒ They can be chained

Promise: Create & Consume

```
const prom = new Promise(
  (resolve, reject) => {
    ...
    resolve(x);
    ...
    reject(y);
    ...
  }
)
```

```
prom
  .then((x) => {
     ...use x...
  } )
  .catch( (y) => {
     ...use y...
} ) ;
```



Chaining Promises

- One of the most important benefits of Promises
- They provide a natural way to express a sequence of asynchronous operations as a linear chain of `then()` invocations
  - without having to nest each operation within the callback of the previous one
    - the "callback hell" seen before
  - If the error handling code is the same for all steps, you can attach `catch()` to the end of the chain
- Important: always return results, otherwise callbacks won't get the result of a previous promise

```
getRepoInfo()
  .then(repo => getIssue(repo))
  .then(issue => getOwner(issue.ownerId))
  .then(owner => sendEmail(owner.email, 'Some text'))
  .catch(e => {
    // just log the error
    console.error(e)
  })
  .finally(_ => logAction());
});
```

Example: Chaining

- Useful, for instance, with I/O API such as `fetch()`, which returns a Promise

```
const status = (response) => {
  if (response.status >= 200 && response.status < 300) {
    return Promise.resolve(response) // static method to return a fulfilled Promise
  }
  return Promise.reject(new Error(response.statusText))
}
const json = (response) => response.json()

fetch('/todos.json’)
  .then(status)
  .then(json)
  .then((data) => { console.log('Request succeeded with JSON response', data) })
.catch((error) => { console.log('Request failed', error) })
```

Promises in Parallel

- What if we want to execute several asynchronous operations in parallel?
- `Promise.all()`
  - takes an array of Promise objects as its input and returns a Promise
  - the returned Promise will be rejected if at least one of the input Promises is rejected
  - otherwise, it will be fulfilled with an array of the fulfillment values for each of the input promises
  - the input array can contain non-Promise values, too: if an element of the array is not a Promise, it is simply copied unchanged into the output array
- `Promise.race()`
  - returns a Promise that is fulfilled or rejected when the first of the Promises in the input array is fulfilled or rejected
  - if there are any non-Promise values in the input array, it simply returns the first one

Simplifying Writing With `async / await`

- ECMAScript 2017 (ES8) introduces two new keywords, `async` and `await`
  - write promise-based asynchronous code that looks like synchronous code
- Prepend the async keyword to any function means that it will return a Promise
- Prepend `await` when calling an async function (or a function returning a Promise) makes the calling code stop until the promise is resolved or rejected

```
const sampleFunction = async () => {
  return 'test'
}
sampleFunction().then(console.log) // This will log 'test'
```

`async` Functions

- The async function declaration defines an asynchronous function
- Asynchronous functions operate in a separate order than the rest of the code (via the event loop), returning an implicit Promise as their result
  - but the syntax and structure of code using async functions looks like standard synchronous functions

```
async function name([param[, param[, ...param]]]) {
  statements
}
```

`await`

- The `await` operator can be used to wait for a Promise. It can only be used inside an async function
- await blocks the code execution within the async function until the Promise is resolved
- When resumed, the value of the await expression is that of the fulfilled Promise
- If the Promise is rejected, the await expression throws the rejected value
  - If the value of the expression following the await operator is not a Promise, it's converted to a resolved Promise

```
returnValue = await expression ;
```

Example: `async / await`

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved’);
    }, 2000);
  });
}
async function asyncCall() {
  console.log('calling’);
  const result = await resolveAfter2Seconds();
  console.log(result);
}

asyncCall();
```

Example: `async / await`

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved’);
    }, 2000);
  });
}
async function asyncCall() {
  console.log('calling’);
  const result = await resolveAfter2Seconds();
  return 'end’;
}

asyncCall().then(console.log);
```

Examples... Before and After

```
const makeRequest = () => {
  return getAPIData()
    .then(data => {
      console.log(data);
      return "done";
    }
  );
}

let res = makeRequest();
```

```
const makeRequest = async () => {
  console.log(await getAPIData());
  return "done";
};

let res = makeRequest();
```

Promises or `async/await`

- If the output of `function2` is dependent on the output of `function1`, use `await`.
- If two functions can be run in parallel, create two different `async` functions and then run them in parallel `Promise.all(promisesArray)`
- Instead of creating huge async functions with many `await asyncFunction()` in it, it is better to create smaller `async functions` (not too much blocking code)
- If your code contains blocking code, it is better to make it an `async` function. The callers can decide on the level of asynchronicity they want.

Async/Await

- Promises are a powerful construct to simplify asynchronous programming
- They are provided as objects and methods

Compare/Contrast

- Promises
  - More specific model of dependent chains of tasks you initiate that will complete later
  - Powerful, flexibly programmable
  - Flexibility can make them confusing

- Async/Await
  - Creates illusion of sequential programming
  - Simplest mental model
  - Need language support
  - Less flexible than promises
  - but often sufficient

Conclusion

- Javascript is the synchronous single-threaded language but with the help of event-loop and promises, JavaScript is used to do asynchronous programming.
- Synchronous means the code runs in a particular sequence of instructions given in the program, whereas asynchronous code execution allows to execution of the upcoming instructions immediately.
- Because of asynchronous programming we can avoid the blocking of tasks due to the previous instructions.
- Both synchronous and asynchronous javascript is useful in different scenarios.

Homework

- Write a function `checkInventory(itemName)` that returns a Promise. The function should resolve with the message "In stock" if `itemName` is 'Laptop', and reject with the message "Out of stock" otherwise. (Simulate the operation using `setTimeout` for 500ms)
- Write code to call the function `checkInventory('Laptop')` and use `.then()` to `console.log` the successful message and `.catch()` to `console.log` any errors.
- Write an async function named `getUserData(userId)` that simulates fetching user data from two APIs sequentially: 1) `fetchUser(userId)` 2) `fetchPosts(user.username)`. It should return an **Object** combining the user and post data. (Assume `fetchUser` and `fetchPosts` return resolved Promises).

Homework

- Convert the following `Promise` Chaining call to `fetchData()` into an `async` function using `await` instead of `.then()`

```
function fetchData() {
  console.log("Fetching Data...");
  return Promise.resolve('Data 1');
}

function processData(data) {
  console.log(`Processing: ${data}`);
  return Promise.resolve(data + ' processed');
}

fetchData()
  .then(data1 => processData(data1))
  .then(finalData => console.log('Result:', finalData));
```