(CS360) Fundamentals of Software Engineering

Async functions

Lecture 4

Dr. Qobiljon Toshnazarov

Outline

New Uzbekistan University

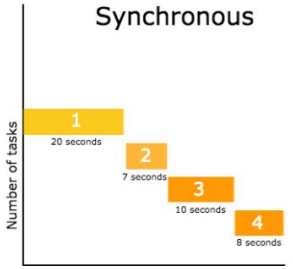
Asynchronous programming:

- Event handler model
- Promise (`.then` and `.catch` properties)
- Async / await

Sync vs. async programming

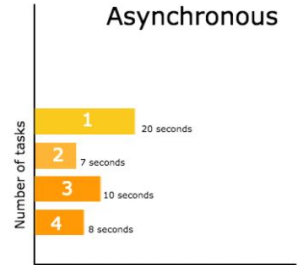
New Uzbekistan University

- **Synchronous programming**
  - Tasks are executed sequentially
  - Blocking execution (operations wait for the previous one to complete)
  - Makes processes are simple and predictable
  - But can be slow for time-consuming (large) operations



With sync. programming, program may handle 4 tasks in 45 seconds overall.

- **Asynchronous programming**
  - Tasks are executed concurrently (i.e., in parallel)
  - Non-blocking execution (+better resource utilization)
  - Can be relatively complex due to use of callbacks, promises, or async/await syntax, etc.
  - Efficient method for performing time-consuming operations



With async. programming, program may handle 4 tasks in 20 seconds overall.

Event handler model

JavaScript runtime environment

New Uzbekistan University

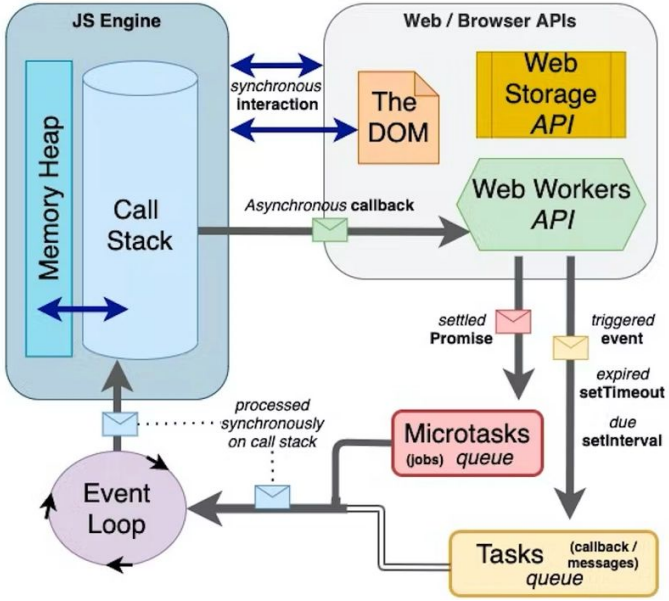
Two pieces of software (in browsers, Node.js, etc.) work together to run your JS code:

JavaScript engine

- Code compilation / parsing
- Code execution

Host environment

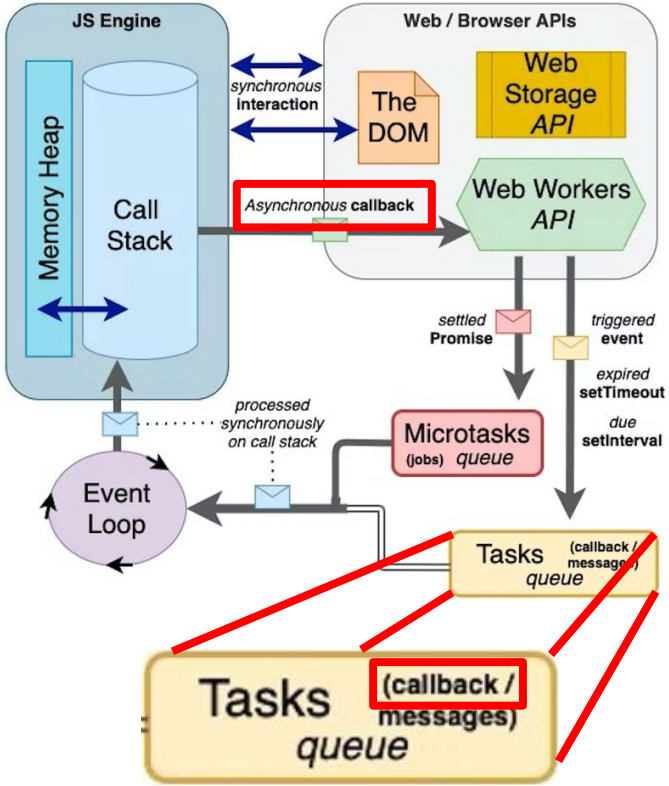
- Provides various APIs specific to a host
- For instance, file system, networking, process management, HTML DOM, etc.



JavaScript runtime environment

New Uzbekistan University

- JS engine asks the host environment to do some work / operation
- JS engine provides an asynchronous callback (function) to be called after host environment is done with the work
- The function is called *"asynchronous"* because JS engine does not have to stop and wait until the work is done
- Callback simply waits to be triggered by the host environment later on, meanwhile JS engine continues doing its own work



JavaScript event handlers

New Uzbekistan University

- An event handler is a function that is waits to be executed when some event happens.
- In Javascript, all the event handlers work in the same address space.
- That means that handlers can communicate through a shared state.
- It also means that switching from one handler to another can be fast.

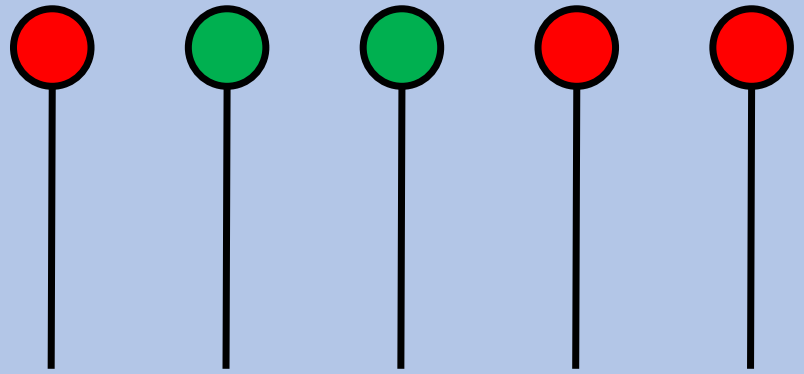
JavaScript event handlers

New Uzbekistan University

The running event handler

One of the event handlers is running; the others are waiting

The pool of waiting event handlers



JavaScript event handler

New Uzbekistan University

- At any time, one event handler is running and the others are waiting.
- Here's an event handler. The color of the head tells us whether it's ready for execution: green if it's ready, red if not.
- This one is not ready: it's still waiting for its event to happen.

JavaScript event handler

New Uzbekistan University

- There are roughly 3 kinds of events that an event handler may be waiting for:
  - Some timer has reached a specific value.
  - Some input/output event occurs.
  - Some other event handler or event handlers complete.

Example: timer event handler

New Uzbekistan University

```
setTimeout(() => {
   console.log(new Date().toLocaleTimeString());
}, 1000);
```

⇨ 3:46:09 PM

```
setTimeout(() => {
   console.log(new Date().toLocaleTimeString());
}, 2000);
```

⇨ 3:46:10 PM

```
setTimeout(() => {
   console.log(new Date().toLocaleTimeString());
}, 3000);
```

⇨ 3:46:11 PM

Event handler semantics

New Uzbekistan University

- JavaScript has "run-to-completion" semantics
  (when an event handler runs, it always runs to completion)
- It is never interrupted.
- This means that a handler doesn't have to worry about some other handler overwriting its memory.
- But this also means that some high-priority task (like responding to a keystroke) can't interrupt a lower-priority task.

Example: `alert()` blocks event loop

New Uzbekistan University

```
setTimeout(() => {
    console.log(new Date().toLocaleTimeString());
    alert("Blocking");
}, 1000);
```

3:53:02 PM

```
setTimeout(() => {
    console.log(new Date().toLocaleTimeString());
}, 2000);
```

3:53:46 PM

```
setTimeout(() => {
    console.log(new Date().toLocaleTimeString());
}, 3000);
```

3:53:46 PM

Event handler semantics

New Uzbekistan University

- So, you want to organize your computation into many handlers, each of which runs to completion quickly.
- This is sometimes called "cooperative multiprocessing".
- The JavaScript programming model is designed to facilitate this, which revolves around concept of “Promises”.

Promise

Promise

New Uzbekistan University

- A *promise* is an object representing the eventual completion or failure of a handler.
- A promise is always in one of three states:
  1. *Fulfilled* – (or resolved) meaning that the handler completed successfully
  2. *Rejected* – meaning that the handler failed
  3. *Pending* – promise is not completed yet (neither fulfilled nor rejected)
- In JS's perspective: promise is either fulfilled or rejected. And once a promise is fulfilled or rejected, it stays that way.

Event handlers as callable objects

New Uzbekistan University

- A promise may have a *.then* property, which is a handler to be invoked when the promise is *fulfilled*

- A promise may also have a *.catch* property, which is a handler to be invoked when the promise is *rejected*

Promises

New Uzbekistan University

- Mostly likely, you will NOT be building promises from scratch - you will probably use a library or a snippet.
- Asynchronous operations (like input/output operations) are typically exported as functions that return promises.
- So, now we concentrate on the use of promises by utilizing the `.then` and `.catch` properties.

Examples for promise

New Uzbekistan University

For our examples, we'll create promises using a function with the following interface:

```
function makePromise1(
   promiseName: string,
   shouldSucceed: boolean,
   value?: number
) : Promise<number>

    // function returns a promise that fulfills with the given value
    // in case shouldSucceed parameter is true, otherwise rejected.
    // 'value' is optional - fulfills with the given value.
```

One possible implementation…

New Uzbekistan University

```
function makePromise1(
    promiseName: string,
    shouldSucceed: boolean,
    value?: number
): Promise<number> {
    console.log(`creating new promise ${promiseName}`)
    return new Promise((resolve, reject) => {
        console.log(`promise ${promiseName} now running; flag = ${shouldSucceed}`)
        setTimeout((): void => {
            if (shouldSucceed) {
                console.log(`promise ${promiseName} now fulfilling with ${value}`)
                resolve(value)
            } else {
                console.log(`promise ${promiseName} now rejecting`)
                reject(`promise ${promiseName} failed`)
            }
        }, 1000)
    })
}
```

makePromise1() in action

New Uzbekistan University

```
import makePromise1 from './promiseMaker'

console.log("main handler starting")

// create a new promise,
// labeled "promise100",
// and throw it in the event pool
let p1 = makePromise1("promise100", true, 10)

// finish the main handler
console.log('main handler finished')
// and go on to run any handlers left in the pool
```

```
main handler starting
creating new promise promise100
main handler finished
promise promise100 now running; flag = true
promise promise100 now fulfilling with 10
```

Extending promises with callbacks

New Uzbekistan University

- Assume that `p1` is a promise that has `.then` property in it.
- `const p2 = p1.then(callback)`
  creates a new promise object that represents the result of promise `p1` followed by the callback (if `p1` fulfills)
- This creates a new promise.

Extending promises with callbacks

New Uzbekistan University

- `const p2 = p1.then(callback)`
- `p2` is ready when `p1` is completed (either fulfilled or rejected)
- When `p2` is pulled from the event queue, the state of `p1` promise is looked up:
  - In case `p1` was fulfilled, its value is passed to the callback, and `p2` completes normally. Note that `p1` does not run again here.

Extending promises with callbacks

New Uzbekistan University

- In case `p1` was rejected, then `p2` exits with an unhandled error.

Linking the event handlers

New Uzbekistan University

```
import makePromise1 from './promiseMaker'

console.log("main handler starting")

const p1 = makePromise1("p1", true, 10)
const p2 = makePromise1("p2", true, 20)
const p3 = p1.then(n => {
    console.log(`p1 passed ${n} to its callbac
})
const p4 = p3.then(() => {
    console.log(`p3 passed no value to its cal
})

console.log("main handler finishing\n")
```

p3 is a new promise that includes both **p1** and the new callback.

```
main handler starting
creating new promise p1
creating new promise p2
main handler finishing

promise p1 now running; flag = true
promise p1 now fulfilling with 10
p1 passed 10 to its callback
p3 passed no value to its callback
promise p2 now running; flag = true
promise p2 now fulfilling with 20
```

`.then` callbacks ignore rejected promises

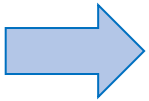
New Uzbekistan University

```
import makePromise1 from './promiseMaker'

console.log("main handler starting")

// p1 will be rejected
const p1 = makePromise1("p1", false, 10)
const p2 = makePromise1("p2", true, 20)

// p3 completes without running the callback (throws an error instead)
const p3 = p1.then(n => {
    console.log(`p1 passed ${n} to its callback`)
})
// and p4 similarly completes without running its
// callback, so it completes with an unhandled exception
const p4 = p3.then(() => {
    console.log(`p3 passed no value to its callback`)
})

console.log("main handler finishing\n")
```

Use a **.catch** callback to catch rejected promises

New Uzbekistan University

```
import makePromise1 from './promiseMaker'

console.log("main handler starting")

// p1 will be rejected
const p1 = makePromise1("p1", false, 10)
const p2 = makePromise1("p2", true, 20)

// p3 throws an error
const p3 = p1.then(n => {
    console.log(`p1 passed ${n} to its callback`)
})
// but p4 catches it
const p4 = p3.catch((e) => {
    console.log(`p3 was rejected; the rejection message was "${e}"`)
})

console.log("main handler finishing\n")
```

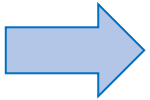


```
main handler starting
creating new promise p1
creating new promise p2
main handler finishing

promise p1 now running; flag = false
promise p1 now rejecting
p3 was rejected; the rejection message
was "promise p1 was rejected"
promise p2 now running; flag = true
promise p2 now fulfilling with 20
```

You can even link more than one callback to a promise

New Uzbekistan University

```
import makePromise1 from './promiseMaker'

console.log("main handler starting")

const p1 = makePromise1("p1", true, 10)
const p2 = makePromise1("p2", true, 20)

const p3 = p1.then(n => {
    console.log(`callback A says: p1 passed ${n} to me`)
})

const p4 = p1.then(n => {
    console.log(`callback B says: p1 passed ${n} to me, too`)
})

console.log("main handler finishing\n")
```

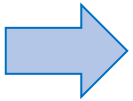


```
main handler starting
creating new promise p1
creating new promise p2
main handler finishing

promise p1 now running; flag = true
promise p1 now fulfilling with 10
callback A says: p1 passed 10 to me
callback B says: p1 passed 10 to me, too
promise p2 now running; flag = true
promise p2 now fulfilling with 20
```

Synchronizing event handlers

New Uzbekistan University

```
import makePromise1 from './promiseMaker'

console.log("main handler starting")

const p1 = makePromise1("p1", true, 10)
const p2 = makePromise1("p2", true, 20)

const p3 = p1.then(n => {
    console.log(`callback A says: p1 passed ${n} to me`);
    return n+1
})

const p4 = p1.then(n => {
    console.log(`callback B says: p1 passed ${n} to me, too`);
    return n+100
})

const p5 = Promise.all([p4,p3])
    .then(values => {
        console.log(`p3 returned ${values[1]}`);
        console.log(`p4 returned ${values[0]}`)
    })

console.log("main handler finishing\n")
```



```
main handler starting
creating new promise p1
creating new promise p2
main handler finishing

promise p1 now running; flag = true
promise p1 now fulfilling with 10
callback A says: p1 passed 10 to me
callback B says: p1 passed 10 to me, too
p3 returned 11
p4 returned 110
promise p2 now running; flag = true
promise p2 now fulfilling with 20
```

Async / await

Promise with **async**

New Uzbekistan University

f1 and **f2** async functions that promise and resolve to a result:

```
// promises a string result
function f1() {
   return Promise.resolve("f1")
}

// or equivalently
async function f2() {
   return "f2"
}
```

f3 and **f4** async functions that also promise but fail to resolve

```
// promises... but throws error
function f3() {
   return Promise.reject("f3 error")
}

// or equivalently
async function f4() {
   throw "f4 error"
}
```

Invoke async with `await` or `.then` & `.catch`

New Uzbekistan University

One way to invoke: just call async functions and get **`Promise`** objects

```
const p1 = f1() // Promise object
const p2 = f2() // Promise object
const p3 = f3() // Promise object
const p4 = f4() // Promise object

p1.then(console.log) // OUT: "f1"
p2.then(console.log) // "f2"
p3.catch(console.log) // "f3 error"
p4.catch(console.log) // "f4 error"
```

Another way using the **await** keyword before function invocation

```
console.log(await f1()) // OUT: "f1"
console.log(await f2()) // "f2"
try {
    await f3()
} catch (e) {
    console.log(e) // "f3 error"
}
try {
    await f4()
} catch (e) {
    console.log(e) // "f4 error"
}
```

Same thing with `.then` & `.catch`

New Uzbekistan University

```
p1.then(console.log, console.log) // "f1", not called
p2.then(console.log, console.log) // "f2", not called
p3.then(console.log, console.log) // not called, "f3 error"
p4.then(console.log, console.log) // not called, "f4 error"
```

```
p1.then(console.log).catch(console.log) // "f1", not called
p2.then(console.log).catch(console.log) // "f2", not called
p3.then(console.log).catch(console.log) // not called, "f3 error"
p4.then(console.log).catch(console.log) // not called, "f4 error"
```

Things to know about async/await

New Uzbekistan University

- An async function always returns a promise.
- Because a promise object is created, it is automatically thrown in the pool of event handlers to be run when ready.
- The async keyword tells the compiler to do the translation into **`.catch`** and **`.then`** (this is specific to JS!)
- Therefore, **`await`** makes no sense except in the body of an async function.
- The try/catch clause is optional.

Long story to reach a simple conclusion

New Uzbekistan University

- A useful but complex pattern of behaviors is encapsulated in a single language construct.
- In the old days, this might have been a "design pattern"
- Illustrates the power of programming-language technology.


Redo
Undrln
Q & A
Find
J
C
V
B
N