CS 4350: Fundamentals of Software Engineering
CS 5500: Foundations of Software Engineering

Lesson 4.1: The Javascript Event Handler Model

Jon Bell, John Boyland, Mitch Wand

Khoury College of Computer Sciences

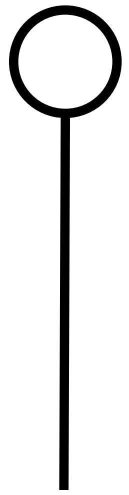
© 2021 Jonathan Bell, John Boyland and Mitch Wand. Released under the CC BY-SA license

Learning Objectives for this Lesson

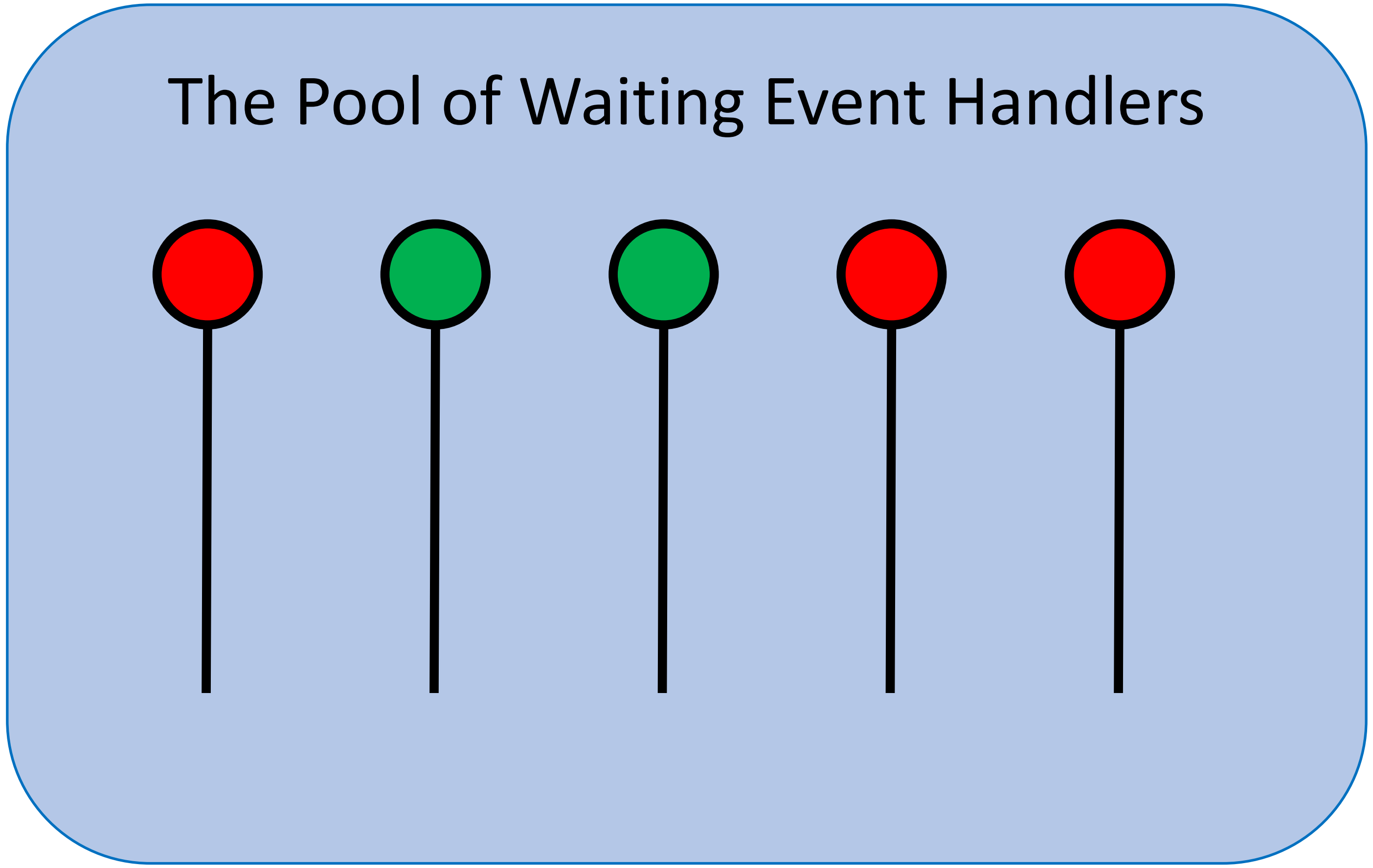
- By the end of this lesson you should be prepared to:
  - Explain what is meant by "run-to-completion semantics"
  - Describe 3 ways in which event handlers may become ready for execution
  - Explain what a "promise" is
  - Given a program consisting of straight-line promises like the ones in the examples, predict the order in which the different pieces can execute.

A JavaScript execution state consists of a bunch of event handlers

The running event handler



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



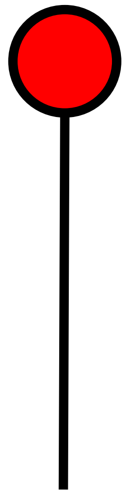
What's an event handler?

- An event handler is a piece of code that is waiting for some event to happen.
- In Javascript, all the event handlers work in the same address space
- That means that handlers can communicate through shared state
- It also means that switching from one handler to another can be fast.

People use different names for these things. Some call them "callbacks"; others call them "messages"; others might call them "green threads". "Event handlers" seems like the best name for now.

The Javascript runtime maintains a pool of event handlers.

- 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.



What's an event?

- 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.

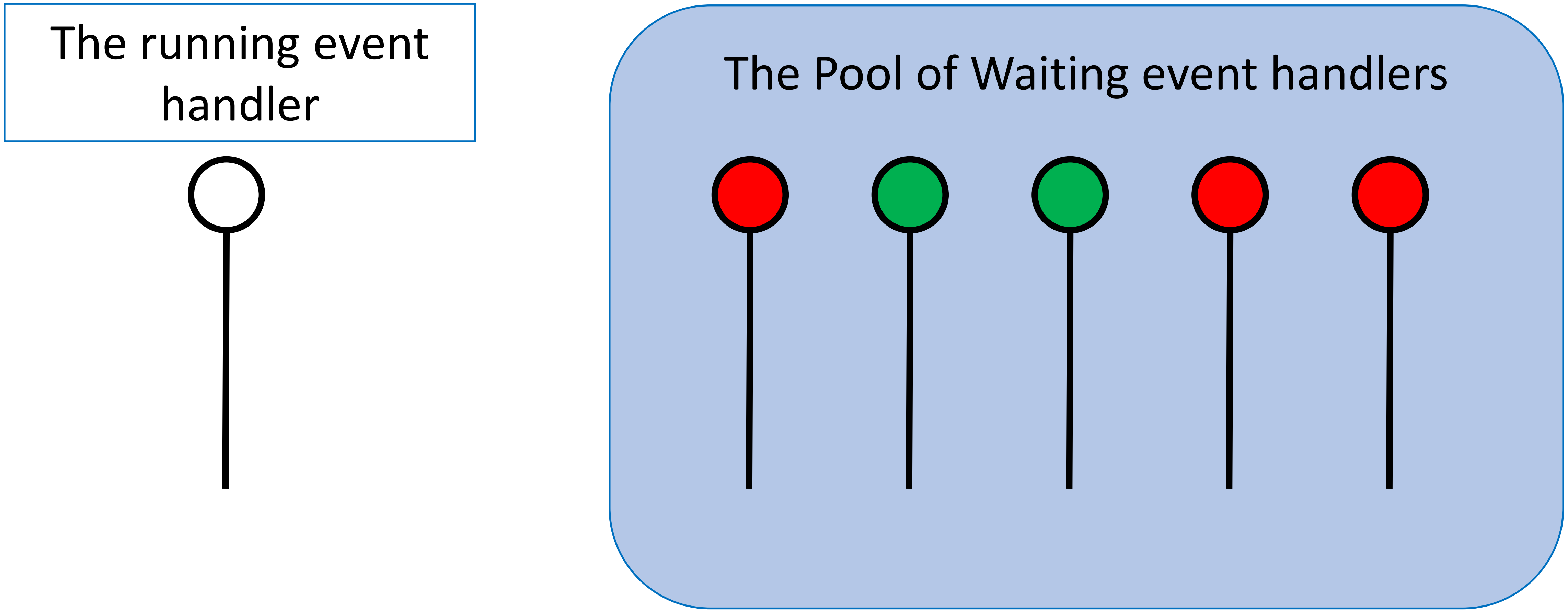
the most common case

our focus in this lesson

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
- 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.

When the running event handler completes, the scheduler chooses one of the other ready event handlers to execute



4.1/example1.ts

How are new event handlers created?

- Simplest way– via JS **setTimeout**

```
console.log("main event handler running")
setTimeout(() => {
    console.log("event handler 2 running")
    console.log("event handler 2 finishing")
})
console.log("main event handler finishing")
```

setTimeout(callback,t) creates a new handler, which runs the callback after a delay of at least t msecs. Default value of t is 0.

- Output:

```
main event handler running
main event handler finishing
event handler 2 running
event handler 2 finishing
```

Handlers as objects

- A *promise* is an object representing the eventual completion or failure of a handler.
- A promise is always in one of three states:
  - *pending*
  - *fulfilled* (or resolved) meaning that the handler completed successfully
  - *rejected*, meaning that the handler failed
- Once a promise is fulfilled or rejected, it stays that way.
- 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

You will most likely not be building promises from scratch

- Asynchronous operations (like input/output operations) are typically exported as functions that return promises.
- So we'll concentrate on the use of promises, by utilizing the .**then** and .**catch** properties.
- For our examples, we'll create promises using a function with the following interface:

```
function makePromise1(promiseName: string, shouldSucceed: boolean, value?: number)
    : Promise<number>
    // returns a promise that fulfills with the given value if shouldSucceed is true
    // and that is rejected otherwise.   'value' is an optional argument
```

4.1/example1a.ts

makePromise1 in action

```
import makePromise1 from './promiseMaker'

console.log("main handler starting")

// create a new promise,
// labeled "promise100",
// and throw it in the pool
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

- const p2 = p1.then(callback) creates a new promise that represents the result of promise p1 followed by the callback (if p1 fulfills)
- This is a *new* promise.
- p2 is ready when p1 is completed (either fulfilled or rejected)
- When p2 is run, it refers to p1. If p1 was fulfilled, its value is passed to the callback, and p2 completes normally. p1 is *not* run again.
- If p1 was rejected, then p2 exits with an unhandled error.

4.1/example2.ts

Linking event handlers

p3 is a new promise that includes both p1 and the new callback.
p4 is a new promise that includes both 3 and the new callback

```
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 callback`)
})
const p4 = p3.then(() => {
    console.log(`p3 passed no value to its callback`)
})

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
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
```

4.1/example3.ts

.**then** callbacks ignore rejected promises

```
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
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

4.1/example4.ts

```
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

```typescript
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
```

When p1 finishes, the callbacks at both p3 and p4 become ready for execution. Their order of execution is unspecified.

4.1/example5.ts

Linking callbacks in series

```
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:number) => {
    console.log(`callback A says: p1 passed ${n} to me`);
    return true
})

const p4 = p3.then((b:boolean) => {
    console.log(`callback B says: callback A passed ${b} to me`)
})


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: callback A passed
true to me
promise p2 now running; flag = true
promise p2 now fulfilling with 20
```

4.1/example6.ts

Synchronizing event handlers

values is bound to the list containing the value passed by p4 and the value passed by p3, in that order, regardless of the order in which p3 and p4 executed.

```javascript
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
```

Review: Learning Objectives for this Lesson

- You should now be able to:
  - Explain what is meant by "run-to-completion semantics"
  - Describe 3 ways in which event handlers may become ready for execution
  - Explain what a "promise" is
  - Given a program consisting of straight-line promises like the ones in the examples, predict the order in which the different pieces can execute.

Next steps...

- Be prepared to explain each line in the output examples
- Create some examples like the ones here and try to predict what they will do.
- Go on to the next lesson