

First-class functions

First-class functions

Functions in JavaScript are objects.

- They can be saved in variables
- They can be passed as parameters
- They have properties, like other objects
- They can be defined without an identifier

(This is also called having first-class functions, i.e. functions in JavaScript are "first-class" because they are treated like any other variable/object.)

Back to the veeeeery basics

What is code?

- A list of instructions your computer can execute
- Each line of code is a statement

What is a function?

- A labeled group of statements
- The statements in a function are executed when the function is invoked

What is a variable?

- A labeled piece of data

Recall: Objects in JS

Objects in JavaScript are sets of property-value pairs:

```
const bear = {  
  name: 'Ice Bear',  
  hobbies: ['knitting', 'cooking', 'dancing']  
};
```

- Like any other value, Objects can be saved in **variables**.
- Objects can be passed as parameters to functions

Back to the veeeeery basics

What is code?

- A list of instructions your computer can execute
- Each line of code is a statement

What is a function?

- A labeled group of statements
- The statements in a function are executed when the function is invoked

What is a variable?

- A labeled piece of data

What could it mean for a function to be an object, i.e. a kind of data?

Function variables

You can declare a function in several ways:

```
function myFunction(params) {  
}
```

```
const myFunction = function(params) {  
};
```

```
const myFunction = (params) => {  
};
```

Function variables

```
function myFunction(params) {  
}
```

```
const myFunction = function(params) {  
};
```

```
const myFunction = (params) => {  
};
```

Functions are invoked in the same way, regardless of how they were declared:

```
myFunction();
```

```
const x = 15;  
let y = true;
```

```
const greeting = function() {  
  console.log('hello, world');  
}
```

"A function in JavaScript is an object of type Function"

➡ `const x = 15;`
`let y = true;`

```
const greeting = function() {  
  console.log('hello, world');  
}
```

"A function in JavaScript is an object of type Function"

In the interpreter's memory:

x

15

```
const x = 15;
```

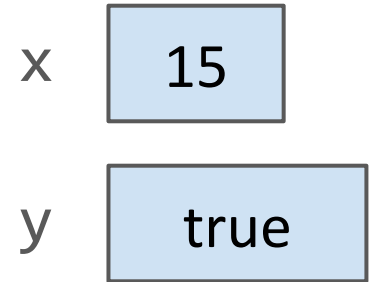
➔

```
let y = true;
```

```
const greeting = function() {  
  console.log('hello, world');  
}
```

"A function in JavaScript is an object of type Function"

In the interpreter's memory:



```
const x = 15;  
let y = true;
```

➔

```
const greeting = function() {  
  console.log('hello, world');  
}
```

"A function in JavaScript is an object of type Function"

In the interpreter's memory:

x 15

y true

greeting ...

```
const x = 15;  
let y = true;
```

```
const greeting = function() {  
  console.log('hello, world');  
}
```



"A function in JavaScript is an object of type Function"

What this really means:

- When you declare a function, there is an object of type Function that gets created alongside the labeled block of executable code.

Function properties

```
const greeting = function() {  
  console.log('hello, world');  
}
```

```
console.log(greeting.name);  
console.log(greeting.toString());
```

When you declare a function, you create an object of type Function, which has properties like:

- name
- toString

Function properties

```
const greeting = function() {  
  console.log('hello, world');  
}
```

```
greeting.call();
```

Function objects also have a call method, which invokes the underlying executable code associated with this function object.

Function properties

```
const greeting = function() {  
  console.log('hello, world');  
}
```

```
greeting.call();  
greeting();
```

- () is an operation on the Function object ([spec](#))
- When you use the () operator on a Function object, it is calling the object's `call()` method, which in turn executes the function's underlying code

Code vs Functions

Important distinction:

- **Function, the executable code**
 - A group of instructions to the computer
- **Function, the object**
 - A JavaScript object, i.e. a set of property-value pairs
 - Function objects have executable code associated with them
 - This executable code can be invoked by
 - *functionName()*; or
 - *functionName.call()*;

Note: Function is special

Only Function objects have executable code associated with them.

- Regular JS objects **cannot** be invoked
- Regular JS objects **cannot** be given executable code
 - I.e. you can't make a regular JS object into a callable function

```
const bear = {  
  name: 'Ice Bear',  
  hobbies: ['knitting', 'cooking', 'dancing']  
};
```

```
bear(); // error!
```

✖ ▶ Uncaught TypeError: bear is not a function

Function Objects vs Objects

```
function sayHello() {  
  console.log('Ice Bear says hello');  
}
```

```
const bear = {  
  name: 'Ice Bear',  
  hobbies: ['knitting', 'cooking', 'dancing'],  
  greeting: sayHello  
};  
bear.greeting();
```

[CodePen](#)

But you can give your object Function properties and then invoke those properties.

Function Objects vs Objects

```
function sayHello() {  
  console.log('Ice Bear says hello');  
}
```

```
const bear = {  
  name: 'Ice Bear',  
  hobbies: ['knitting', 'cooking', 'dancing'],  
  greeting: sayHello  
};  
bear.greeting();
```

[CodePen](#)

The **greeting** property is an object of Function type.

Why do we have Function objects?!

Callbacks

Function objects **really** come in handy for event-driven programming!

```
function onDragStart(event) {  
    ...  
}  
dragon.addEventListener('pointerdown', onDragStart);
```

Because every function declaration creates a Function object, we can pass Functions as parameters to other functions.

Creating functions within functions

Functions that create functions

In JavaScript, we can **create** functions from within functions ([CodePen](#)).

```
function printMessage(birthYear) {  
  function getLabel(age) {  
    if (age < 2) {  
      return "baby";  
    }  
    if (age < 4) {  
      return "toddler";  
    }  
    if (age < 13) {  
      return "kid";  
    }  
    if (age < 20) {  
      return "teenager";  
    }  
    return "grown-up";  
  }  
  
  const ageThisYear = 2017 - birthYear;  
  const label = getLabel(ageThisYear);  
  console.log('You are a ' + label + ' this year.');
```

`printMessage(2005);`

Functions that create functions

In JavaScript, we can **create** functions from within functions ([CodePen](#)).

```
function printMessage(birthYear) {  
  function getLabel(age) {  
    if (age < 2) {  
      return "baby";  
    }  
    if (age < 4) {  
      return "toddler";  
    }  
    if (age < 13) {  
      return "kid";  
    }  
    if (age < 20) {  
      return "teenager";  
    }  
    return "grown-up";  
  }  
}
```

A function declared within a function is also known as a **closure**.

Scope of closures

```
function printMessage(birthYear) {  
  if (true) {  
    function getLabel(age) {  
      if (age < 2) {  
        return "baby";  
      }  
      if (age < 4) {  
        return "toddler";  
      }  
      if (age < 13) {  
        return "kid";  
      }  
      if (age < 20) {  
        return "teenager";  
      }  
      return "grown-up";  
    }  
  }  
}  
  
const ageThisYear = 2017 - birthYear;  
const label = getLabel(ageThisYear);  
console.log('You are a ' + label + ' this year.');
```

Functions declared with `function` (or `var`) have function scope.

- Can be referenced anywhere in the function after declaration

This example works:

Console

"You are a kid this year."

Scope of closures

```
function printMessage(birthYear) {  
  function getLabel(age) {  
    if (age < 2) {  
      return "baby";  
    }  
    if (age < 4) {  
      return "toddler";  
    }  
    if (age < 13) {  
      return "kid";  
    }  
    if (age < 20) {  
      return "teenager";  
    }  
    return "grown-up";  
  }  
  
  const ageThisYear = 2017 - birthYear;  
  const label = getLabel(ageThisYear);  
  console.log('You are a ' + label + ' this year.');
```

`printMessage(2005):`
`const label = getLabel(8);`

Functions declared with function (or var) have function scope.

- Cannot be referenced outside the function

This example doesn't work:

Scope of closures

```
function printMessage(birthYear) {  
  function getLabel(age) {  
    if (age < 2) {  
      return "baby";  
    }  
    if (age < 4) {  
      return "toddler";  
    }  
    if (age < 13) {  
      return "kid";  
    }  
    if (age < 20) {  
      return "teenager";  
    }  
    return "grown-up";  
  }  
}
```

```
const ageThisYear = 2005  
const label = getLabel(ageThisYear)  
console.log('You are a ' + label + ' this year.')  
}
```

```
printMessage(2005):  
const label = getLabel(8)
```

Functions declared with function (or var) have function scope.

- Cannot be referenced outside the function

This example doesn't work:



top



Filter

Info

You are a kid this year.

✖ Uncaught ReferenceError: getLabel is not defined
at 72165567caf5acb78997480f59e315c6:59

Scope of closures

```
function printMessage(birthYear) {  
  if (true) {  
    const getLabel = function(age) {  
      if (age < 2) {  
        return "baby";  
      }  
      if (age < 4) {  
        return "toddler";  
      }  
      if (age < 13) {  
        return "kid";  
      }  
      if (age < 20) {  
        return "teenager";  
      }  
      return "grown-up";  
    }  
  }  
}
```

Functions declared with **const** or **let** have block scope

- Cannot be referenced outside of the block.

This example doesn't work:

```
const ageThisYear = 2017 - birthYear;  
const label = getLabel(ageThisYear);  
console.log('You are a ' + label + ' this year.');
```

✖ ▶ Uncaught ReferenceError: getLabel is not defined
at printMessage (pen.js:22)
at pen.js:26

Functions that return functions

In JavaScript, we can **return** new functions as well.
(We kind of knew this already because `bind` returns a new function.)

```
function makeHelloFunction(name) {  
  const greeting = function() {  
    console.log('Hello, ' + name);  
  };  
  return greeting;  
}  
  
const helloWorld = makeHelloFunction('world');  
const hello3 = makeHelloFunction('hello, hello');  
  
helloWorld();  
hello3();
```

[CodePen](#)

Functions that create functions

```
function makeHelloFunction(name) {  
  const greeting = function() {  
    console.log('Hello, ' + name);  
  };  
  return greeting;  
}
```

```
const helloWorld = makeHelloFunction('world');  
const hello3 = makeHelloFunction('hello, hello');
```

```
helloWorld();  
hello3();
```

[CodePen](#)



top



Filter

Hello, world

Hello, hello, hello

Closure: an inner function

```
function makeHelloFunction(name) {  
  const greeting = function() {  
    console.log('Hello, ' + name);  
  };  
  return greeting;  
}
```

- When you declare a function inside another function, the inner function is called a **closure**.

Closure: an inner function

```
function makeHelloFunction(name) {  
  const greeting = function() {  
    console.log('Hello, ' + name);  
  };  
  return greeting;  
}
```

- Within a closure, you can reference variables that were declared in the outer function, and those variables **will not go away** after the outer function returns.

Functions that create functions

```
function makeHelloFunction(name) {  
  const greeting = function() {  
    console.log('Hello, ' + name);  
  };  
  return greeting;  
}  
  
const helloWorld = makeHelloFunction('world');  
const hello3 = makeHelloFunction('hello, hello');  
  
helloWorld();  
hello3();
```

The scope of `greeting` is only in the `makeHelloFunction` function, as well as the scope of `name`...

Functions that create functions

```
function makeHelloFunction(name) {  
  const greeting = function() {  
    console.log('Hello, ' + name);  
  };  
  return greeting;  
}  
  
const helloWorld = makeHelloFunction('world');  
const hello3 = makeHelloFunction('hello, hello');  
  
helloWorld();  
hello3();
```

But the makeHelloFunction function returns a reference to the function, which is an object, so the function object doesn't go away

Functions that create functions

```
function makeHelloFunction(name) {  
  const greeting = function() {  
    console.log('Hello, ' + name);  
  };  
  return greeting;  
}  
  
const helloWorld = makeHelloFunction('world');  
const hello3 = makeHelloFunction('hello, hello');  
  
→ helloWorld();  
hello3();
```

And the function object keeps a reference to the name parameter, so that when the created function is called...

Functions that create functions

```
function makeHelloFunction(name) {  
  const greeting = function() {  
    console.log('Hello, ' + name);  
  };  
  return greeting;  
}
```

```
const helloWorld = makeHelloFunction('world');  
const hello3 = makeHelloFunction('3');
```

→

```
helloWorld();  
hello3();
```



top



Filter

Hello, world

... we see that the new function returned from `makeHelloFunction` still has access to the `name` variable.

Functions that create functions

```
function makeHelloFunction(name) {  
  const greeting = function() {  
    console.log('Hello, ' + name);  
  };  
  return greeting;  
}
```

```
const helloWorld = makeHelloFunction('world');  
const hello3 = makeHelloFunction('hello, hello');  
  
helloWorld();  
hello3();
```

The idea of constructing a new function that is "partially instantiated" with arguments is called **currying**. ([article](#))

Anonymous functions

We do not need to give an identifier to functions.

When we define a function without an identifier, we call it an **anonymous function**

- Also known as a **function literal**, or a **lambda function**

```
function makeHelloFunction(name) {  
  const greeting = function() {  
    console.log('Hello, ' + name);  
  };  
  return greeting;  
}
```

Anonymous functions

We do not need to give an identifier to functions.

When we define a function without an identifier, we call it an **anonymous function**

- Also known as a **function literal**, or a **lambda function**

```
function makeHelloFunction(name) {  
  return function() {  
    console.log('Hello, ' + name);  
  };  
}
```

[CodePen](#)

Gotchas and style notes

Recall: Present example

```
class Present {  
  constructor(containerElement, giftSrc) {  
    this.containerElement = containerElement;  
    this.giftSrc = giftSrc;  
  
    this._openPresent = this._openPresent.bind(this);  
  
    const image = document.createElement('img');  
    image.src = OUTSIDE_IMAGE_URL;  
    image.addEventListener('click', this._openPresent);  
    this.containerElement.append(image);  
  }  
  
  _openPresent(event) {  
    const image = event.currentTarget;  
    image.src = this.giftSrc;  
  }  
}
```

We implemented a Present class that had a separate **openPresent** method.

[CodePen](#)

```
class Present {  
  constructor(containerElement, giftSrc) {  
    this.containerElement = containerElement;  
    this.giftSrc = giftSrc;  
  
    const image = document.createElement('img');  
    image.src = OUTSIDE_IMAGE_URL;  
    image.addEventListener('click', function(event) {  
      const image = event.currentTarget;  
      image.src = this.giftSrc;  
    });  
    this.containerElement.append(image);  
  }  
}
```

What would happen if we defined the click event handler directly in the call to `addEventListener` ([CodePen](#))?

```
class Present {  
  constructor(containerElement, giftSrc) {  
    this.containerElement = containerElement;  
    this.giftSrc = giftSrc;  
  
    const image = document.createElement('img');  
    image.src = OUTSIDE_IMAGE_URL;  
    image.addEventListener('click', function(event) {  
      const image = event.currentTarget;  
      image.src = this.giftSrc;  
    });  
    this.containerElement.append(image);  
  }  
}
```



We didn't bind this, so we have a bug:
this is the img instead of the Present object.

```
class Present {  
  constructor(containerElement, giftSrc) {  
    this.containerElement = containerElement;  
    this.giftSrc = giftSrc;  
  
    const image = document.createElement('img');  
    image.src = OUTSIDE_IMAGE_URL;  
    image.addEventListener('click', (function(event) {  
      const image = event.currentTarget;  
      image.src = this.giftSrc;  
    }).bind(this));  
    this.containerElement.append(image);  
  }  
}
```

[Fixed CodePen](#)

```
class Present {  
  constructor(containerElement, giftSrc) {  
    this.containerElement = containerElement;  
    this.giftSrc = giftSrc;  
  
    const image = document.createElement('img');  
    image.src = OUTSIDE_IMAGE_URL;  
    image.addEventListener('click', (function(event) {  
      const image = event.currentTarget;  
      image.src = this.giftSrc;  
    }).bind(this));  
    this.containerElement.append(image);  
  }  
}
```

[Fixed CodePen](#)

```
class Present {  
  constructor(containerElement, giftSrc) {  
    this.containerElement = containerElement;  
    this.giftSrc = giftSrc;  
  
    const image = document.createElement('img');  
    image.src = OUTSIDE_IMAGE_URL;  
    image.addEventListener('click', event => {  
      const image = event.currentTarget;  
      image.src = this.giftSrc;  
    });  
    this.containerElement.append(image);  
  }  
}
```

What would happen if we defined the click event handler like this, with the arrow function instead ([CodePen](#))?

This works! **Why?!**
([CodePen](#))

```
image.addEventListener('click', event => {  
  const image = event.currentTarget;  
  image.src = this.giftSrc;  
});
```



=> versus function

When you define a function using **function syntax**:

```
const onClick = function() {  
    const image = event.currentTarget;  
    image.src = this.giftSrc;  
};
```

this is will be dynamically assigned to a different value depending on how the function is called, like we've seen before (unless explicitly bound with `bind`)

=> versus function

When you define a function using **arrow syntax**:

```
const onClick = event => {  
    const image = event.currentTarget;  
    image.src = this.giftSrc;  
};
```

this is **bound** to the value of **this** in its enclosing context

```
class Present {  
  constructor(containerElement, giftSrc) {  
    this.containerElement = containerElement;  
    this.giftSrc = giftSrc;  
  
    const image = document.createElement('img');  
    image.src = OUTSIDE_IMAGE_URL;  
    image.addEventListener('click', event => {  
      const image = event.currentTarget;  
      image.src = this.giftSrc;  
    });  
    this.containerElement.append(image);  
  }  
}
```

Since we've used the arrow function in the constructor, the **this** in the enclosing context is the new Present object.

Which is better style?

```
class Present {  
  constructor(containerElement, giftSrc) {  
    this.containerElement = containerElement;  
    this.giftSrc = giftSrc;  
  
    this._openPresent = this._openPresent.bind(this);  
  
    const image = document.createElement('img');  
    image.src = OUTSIDE_IMAGE_URL;  
    image.addEventListener('click', this._openPresent);  
    this.containerElement.append(image);  
  }  
  
  _openPresent(event) {  
    const image = event.currentTarget;  
    image.src = this.giftSrc;  
  }  
}
```

(A) Explicit event handler

```
class Present {  
  constructor(containerElement, giftSrc) {  
    this.containerElement = containerElement;  
    this.giftSrc = giftSrc;  
  
    const image = document.createElement('img');  
    image.src = OUTSIDE_IMAGE_URL;  
    image.addEventListener('click', event => {  
      const image = event.currentTarget;  
      image.src = this.giftSrc;  
    });  
    this.containerElement.append(image);  
  }  
}
```

(B) Inline event handler

Callback style

```
image.addEventListener('click', this._openPresent);
```

Version A: Explicit event handler

- Pros:
 - Easier to read
 - More modular
 - Scales better to long functions, several event handlers
- Cons:
 - Because all class methods are public, it exposes the onClick function (which should be private)

Callback style

```
image.addEventListener('click', this._openPresent);
```

Version A: Explicit event handler

- Pros:
 - Easier to read
 - More modular
 - Scales better to long functions, several event handlers
- Cons:
 - Because all class methods are public, it exposes the onClick function (which should be private)
 - Need to bind explicitly

Callback style

```
image.addEventListener('click', this._openPresent);
```

Version A: Explicit event handler

- Pros:
 - Easier to read
 - More modular
 - Scales better to long functions, several event handlers
- Cons:
 - Because all class methods are public, it exposes the onClick function (which should be private)
 - Need to bind explicitly

Callback style

```
image.addEventListener('click', event => {  
  const image = event.currentTarget;  
  image.src = this.giftSrc;  
});
```

Version B: Inline event handler

- Pros:
 - Does not expose the event handler: function is privately encapsulated
- Cons:
 - Constructor logic has unrelated logic inside of it
 - Will get messy with lots of event handlers, long event handlers

Callback style

```
image.addEventListener('click', event => {  
  const image = event.currentTarget;  
  image.src = this.giftSrc;  
});
```

Version B: Inline event handler

- Pros:
 - Does not expose the event handler: function is privately encapsulated
- Cons:
 - Constructor logic has unrelated logic inside of it
 - Will get messy with lots of event handlers, long event handlers

Advanced closures

```
function createFunction() {  
  let x = 0;  
  
  function inner() {  
    x++;  
    let y = 0;  
    y++;  
  
    console.log('x is: ' + x + ', ' + 'y is: ' + y);  
  }  
  return inner;  
}  
  
const functionOne = createFunction();  
functionOne();  
functionOne();  
functionOne();
```

What's the output of this program? ([CodePen](#))

Advanced closures

```
function createFunction() {  
  let x = 0;  
  
  function inner() {  
    x++;  
    let y = 0;  
    y++;  
  
    console.log('x is: ' + x + ', ' + 'y is: ' + y);  
  }  
  return inner;  
}  
  
const functionOne = createFunction();  
functionOne();  
functionOne();  
functionOne();
```

Console

"x is: 1, y is: 1"

"x is: 2, y is: 1"

"x is: 3, y is: 1"

Closures

```
function createFunction() {  
  let x = 0;  
  
  function inner() {  
    x++;  
    let y = 0;  
    y++;  
  
    console.log('x is: ' + x + ', ' + 'y is: ' + y);  
  }  
  return inner;  
}
```

Within a closure, you can reference variables that were declared in the outer function, and those variables **will not go away** after the outer function returns.

Closures

```
function createFunction() {  
  let x = 0;  
  
  function inner() {  
    x++;  
    let y = 0;  
    y++;  
  
    console.log('x is: ' + x + ', ' + 'y is: ' + y);  
  }  
  return inner;  
}
```

The variable is not copied to the inner function; the inner function has a **reference** to the variable in the outer scope.

- [See this iconic StackOverflow post](#) to learn more

Closures

```
function createFunction() {  
  let x = 0;  
  
  function inner() {  
    x++;  
    let y = 0;  
    y++;  
  
    console.log('x is: ' + x + ', ' + 'y is: ' + y);  
  }  
  return inner;  
}
```

tl;dr: Be careful with closures! For now, we are not going to be modifying outer function variables in the closure.

Private fields in object — 2019
What before?

IIFE — Immediately Invoked Function Expression

```
(function(){ console.log("Hello") })()
```

Local variables inaccessible from outside

Can set event listeners that close over variables

JavaScript event loop

Note: see talk!

(For a perfectly great talk on this, see Philip Roberts' talk:
<https://www.youtube.com/watch?v=8aGhZQkoFbQ&t=1s>

And for a perfectly great deep dive on this, see Jake
Archibald's blog post:
[https://jakearchibald.com/2015/tasks-microtasks-queues-a
nd-schedules/](https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/)

These slides are inspired by these resources!)

setTimeout

To help us understand the event loop better, let's learn about a new command, [setTimeout](#):

`setTimeout(function, delay);`

- *function* will fire after *delay* milliseconds
- [CodePen example](#)

Call stack + setTimeout

Call Stack



```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

(global function)

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}
```



```
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

(global function)

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}
```



```
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

`console.log('Point A');`

(global function)

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```



(global function)

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

setTimeout(...);

(global function)

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```



(global function)

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}
```



```
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

`console.log('Point B');`

(global function)

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```



(global function)

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  → console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

onTimerDone()

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  → console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

console.log('Point C');

onTimerDone()

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  → const h1 = document.querySelector('h1');  
    h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

onTimerDone()

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  → const h1 = document.querySelector('h1');  
    h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

querySelector('h1');

onTimerDone()

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  → h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

onTimerDone()

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
} →  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

onTimerDone()

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

Call stack + setTimeout

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

What "enqueues" onTimerDone?
How does it get fired?

Call Stack

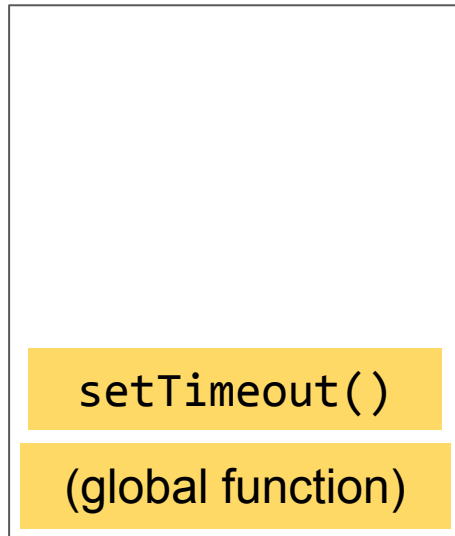
setTimeout(...);

(global function)

Tasks, Micro-tasks, and the Event Loop

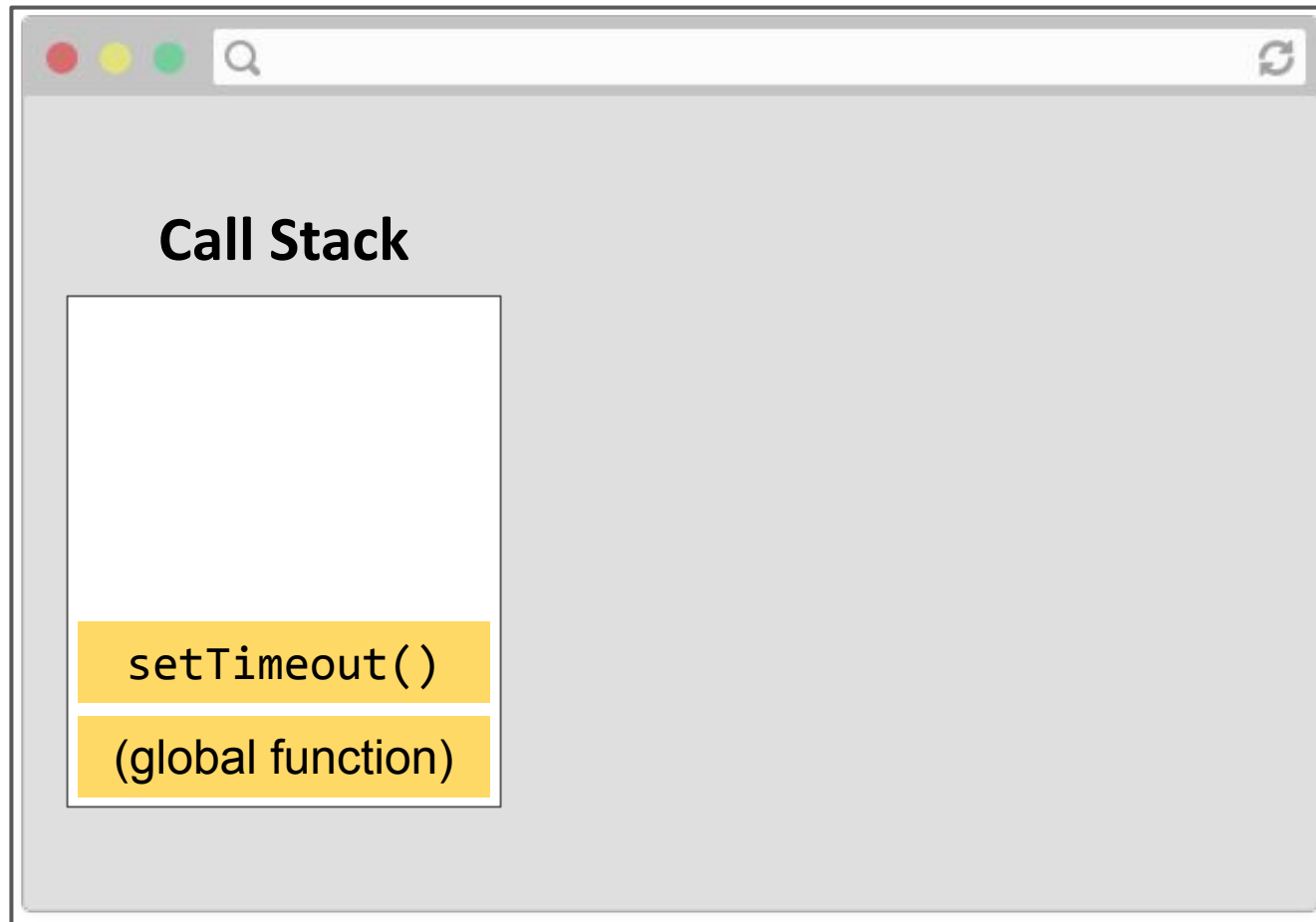
Tasks and the Event Loop

Call Stack

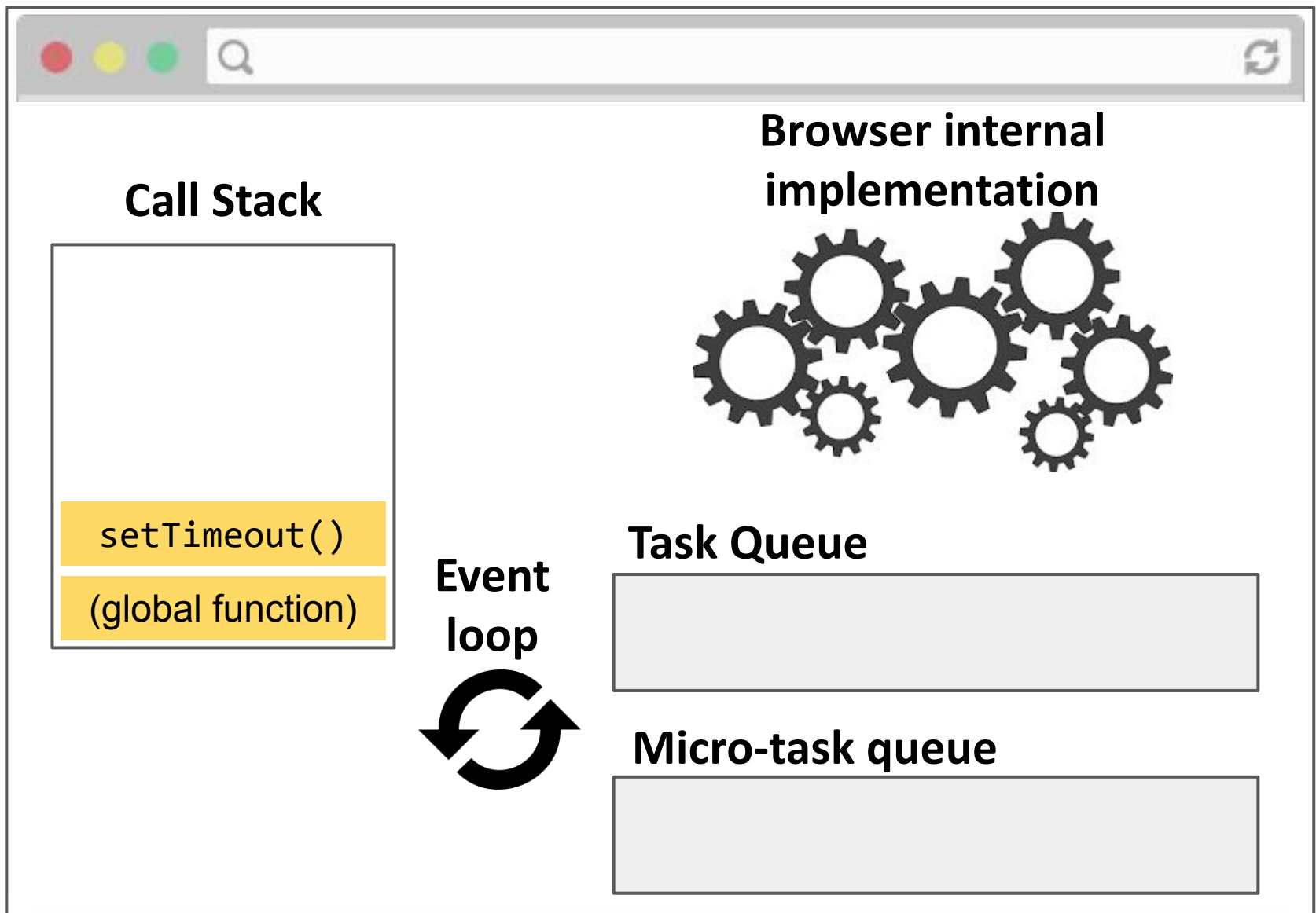


The JavaScript runtime can do only one thing at a time...

Tasks and the Event Loop

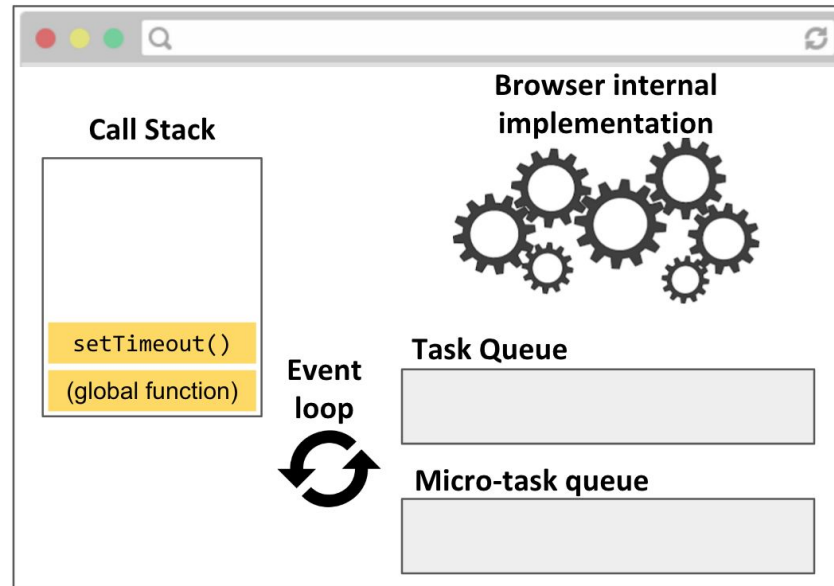


But the JS runtime runs within a browser, which can do multiple things at a time.



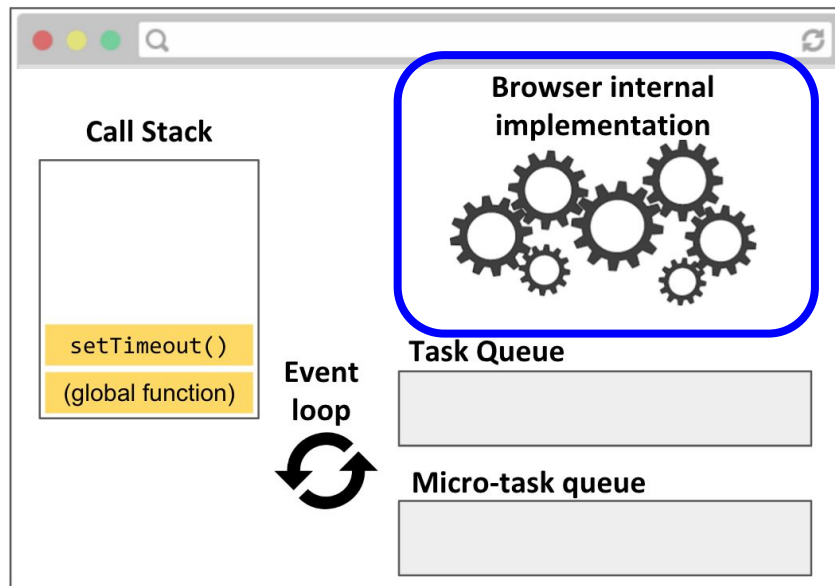
Here's a picture of the major pieces involved in executing JavaScript code in the browser.

JS execution



- **Call stack:** JavaScript runtime call stack. Executes the JavaScript commands, functions.
- **Browser internal implementation:** The C++ code that executes in response to native JavaScript commands, e.g. `setTimeout`, `element.classList.add('style')`, etc.

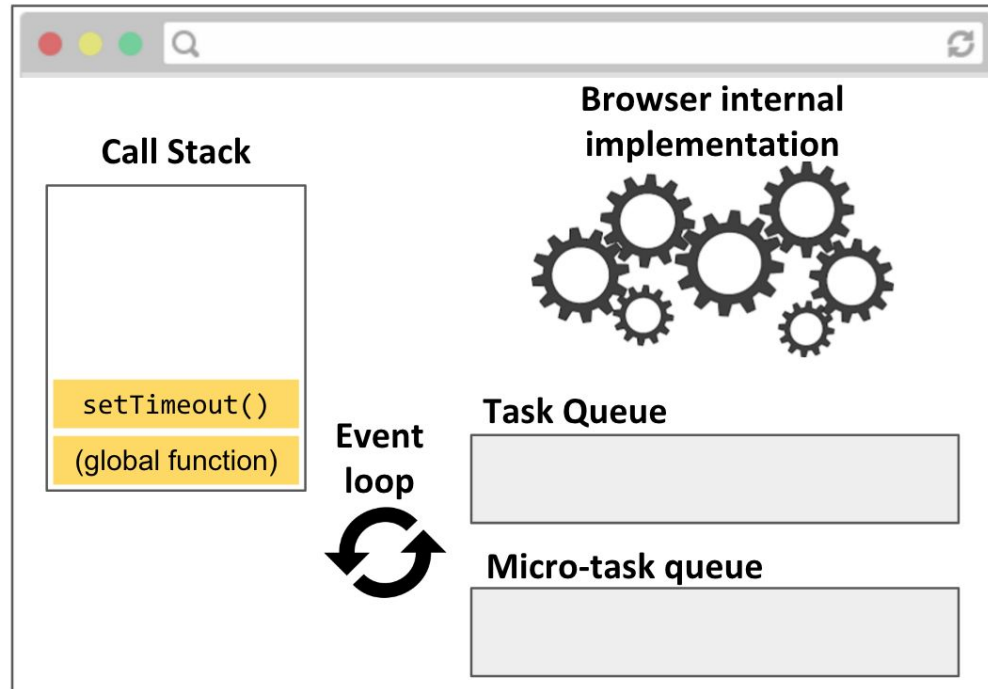
JS execution



The browser itself is multi-threaded and multi-process!

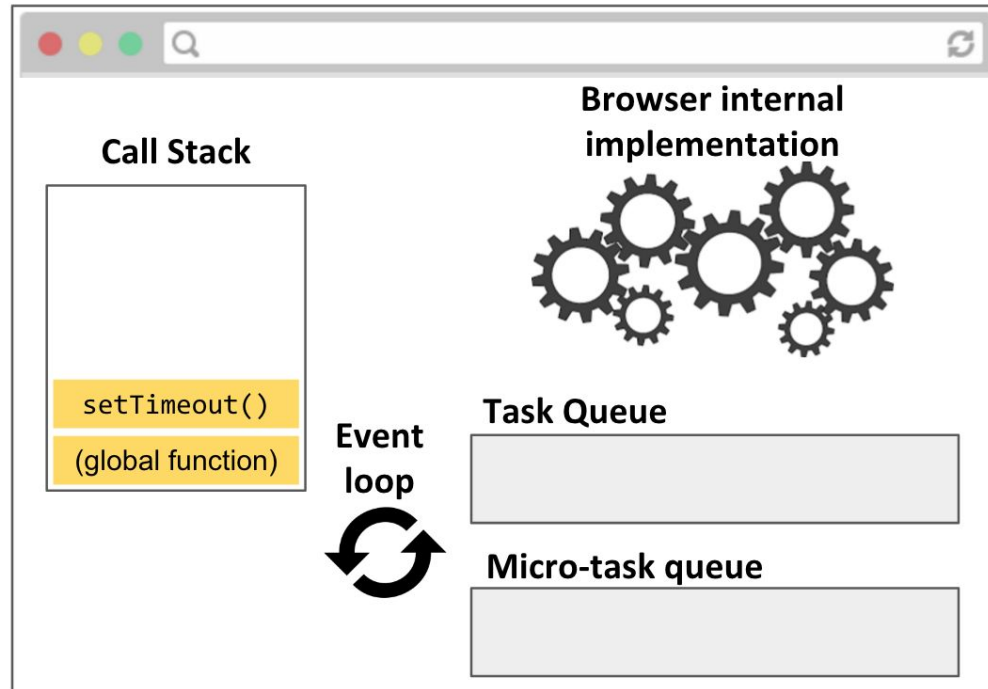
- **Call stack:** JavaScript runtime call stack. Executes the JavaScript commands, functions.
- **Browser internal implementation:** The C++ code that executes in response to native JavaScript commands, e.g. `setTimeout`, `element.classList.add('style')`, etc.

JS execution



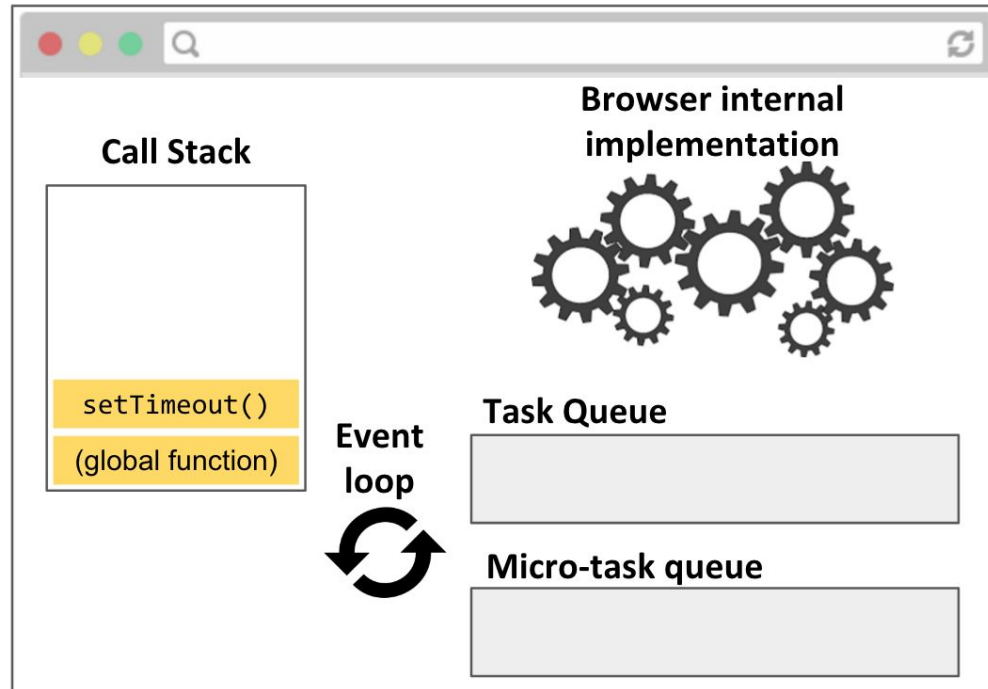
- **Task Queue:** When the browser internal implementation notices a callback from something like `setTimeout` or `addEventListener` is should be fired, it creates a Task and enqueues it in the Task Queue

JS execution



- **Micro-task Queue:** Promises are special tasks that execute with higher priority than normal tasks, so they have their own special queue. ([see details here](#))

JS execution



Event loop: Processes the task queues.

- When the call stack is empty, the event loop pulls the next task from the task queues and puts it on the call stack.
- The Micro-task queue has higher priority than the Task Queue.

Fetch API

Fetch API: `fetch()`

The [Fetch API](#) is the API to use to load external resources (text, JSON, etc) in the browser.

The Fetch API is made up of one function, and its syntax is is concise and easy to use:

```
fetch( 'images.txt' );
```

Note: [XMLHttpRequest](#) ("XHR") is the old API for loading resources from the browser. XHR still works, but is clunky and harder to use.

Fetch API: `fetch()`

The [Fetch API](#) is the API to use to load external resources (text, JSON, etc) in the browser.

The Fetch API is made up of one function, and its syntax is is concise and easy to use:

```
fetch('images.txt');
```

- The `fetch()` method takes the string path to the resource you want to fetch as a parameter
- It returns a `Promise`

Fetch API: `fetch()`

The [Fetch API](#) is the API to use to load external resources (text, JSON, etc) in the browser.

The Fetch API is made up of one function, and its syntax is is concise and easy to use:

```
fetch('images.txt');
```

- The `fetch()` method takes the string path to the resource you want to fetch as a parameter
- It returns a `Promise`
 - **What the heck is a `Promise`?**

Promises:
Another conceptual odyssey

Promises and .then()

A Promise:

- An object used to manage asynchronous results
- Has a `then()` method that lets you attach functions to execute onSuccess or onError
- Allows you to build **chains** of asynchronous results.

Promises are easier to **use** than to **define**...

Simple example: getUserMedia

There is an API called `getUserMedia` that allows you get the media stream from your webcam.

There are two versions of `getUserMedia`:

- `navigator.getUserMedia` (deprecated)
 - Uses callbacks
- `navigator.mediaDevices.getUserMedia`
 - Returns a Promise

getUserMedia with callbacks

```
const video = document.querySelector('video');

function onCameraOpen(stream) {
  video.srcObject = stream;
}

function onError(error) {
  console.log(error);
}

navigator.getUserMedia({ video: true },
  onCameraOpen, onError);
```

[CodePen](#)

getUserMedia with Promise

```
const video = document.querySelector('video');

function onCameraOpen(stream) {
  video.srcObject = stream;
}

function onError(error) {
  console.log(error);
}

navigator.mediaDevices.getUserMedia({ video: true })
  .then(onCameraOpen, onError);
```

[CodePen](#)

Hypothetical Fetch API

```
// FAKE HYPOTHETICAL API.  
// This is not how fetch is called!  
function onSuccess(response) {  
    ...  
}  
function onFail(response) {  
    ...  
}  
fetch('images.txt', onSuccess, onFail);
```

Real Fetch API

```
function onSuccess(response) {  
    ...  
}  
function onFail(response) {  
    ...  
}  
fetch('images.txt').then(onSuccess, onFail);
```

Promise syntax

Q: How does this syntax work?

```
fetch('images.txt').then(onSuccess, onFail);
```

Promise syntax

Q: How does this syntax work?

```
fetch('images.txt').then(onSuccess, onFail);
```

The syntax above is the same as:

```
const promise = fetch('images.txt');  
promise.then(onSuccess, onFail);
```

Promise syntax

```
const promise = fetch('images.txt');  
promise.then(onSuccess, onFail);
```

The object `fetch` returns is of type Promise.

A promise is in one of three states:

- **pending**: initial state, not fulfilled or rejected.
- **fulfilled**: the operation completed successfully.
- **rejected**: the operation failed.

You attach handlers to the promise via `.then()`

Promise syntax

```
const promise = fetch('images.txt');  
promise.then(onSuccess, onFail);
```

The object `fetch` returns is of type Promise.

A promise is in one of three states:

- **pending**: initial state, not fulfilled or rejected.
- **fulfilled**: the operation completed successfully.
- **rejected**: the operation failed.

You attach handlers to the promise via `.then()`

Using Fetch

```
function onSuccess(response) {  
    console.log(response.status);  
}  
fetch('images.txt').then(onSuccess);
```

The success function for Fetch gets a response parameter:

- **response.status**: Contains the status code for the request, e.g. 200 for HTTP success
 - [HTTP status codes](#)

Fetch attempt

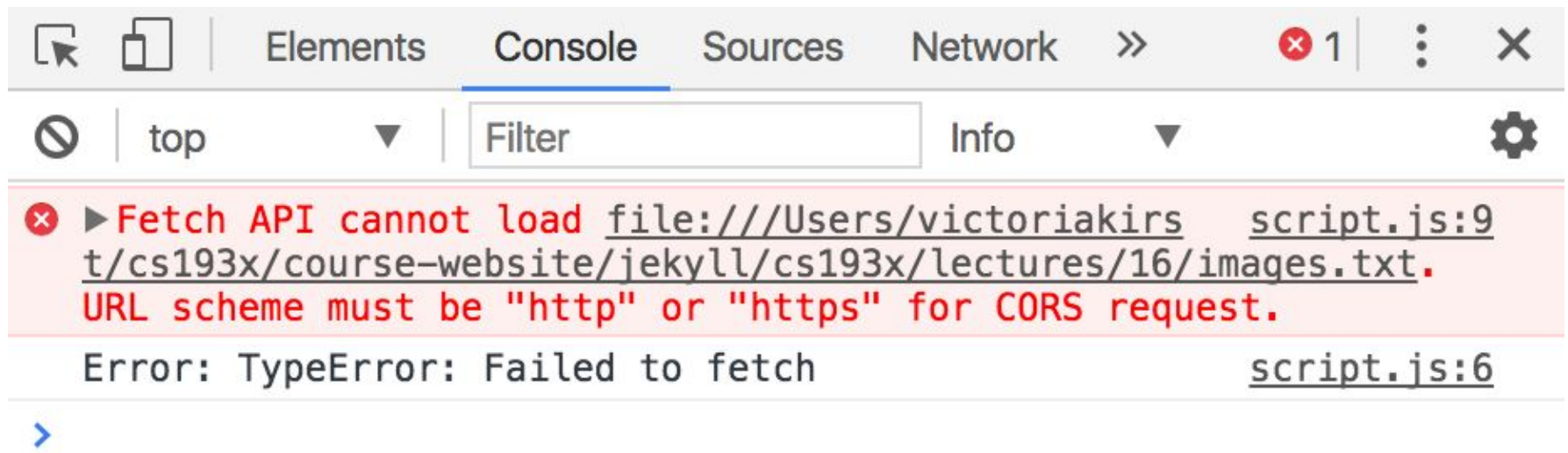
```
function onSuccess(response) {  
    console.log(response.status);  
}
```

```
function onError(error) {  
    console.log('Error: ' + error);  
}
```

```
fetch('images.txt')  
    .then(onSuccess, onError);
```

Fetch error

If we try to load this in the browser, we get the following JavaScript error:



Notice that our `onError` function was also called.

Local files

When we load a web page in the browser that is saved on our computer, it is served via `file://` protocol:



We are **not allowed** to load files in JavaScript from the `file://` protocol, which is why we got the error.

Serve over HTTP

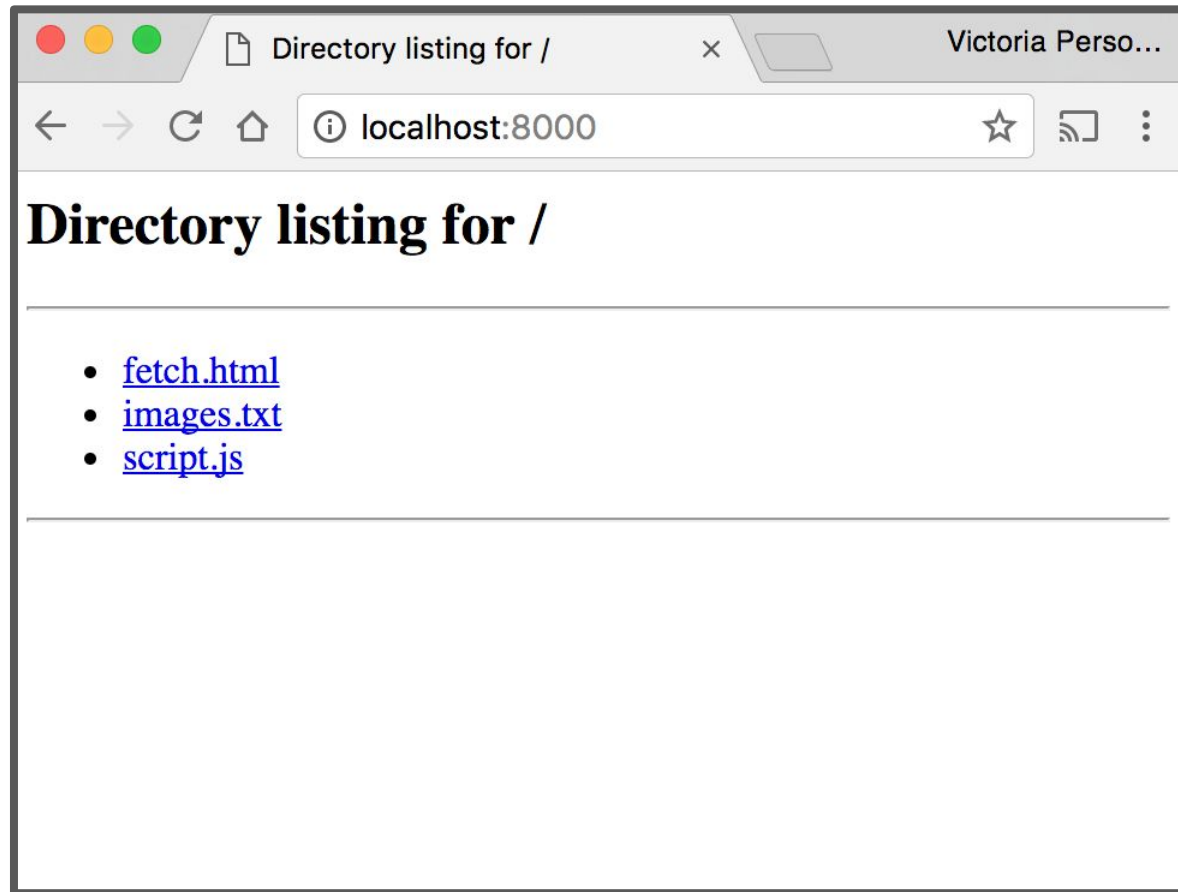
We can run a program to serve our local files over HTTP:

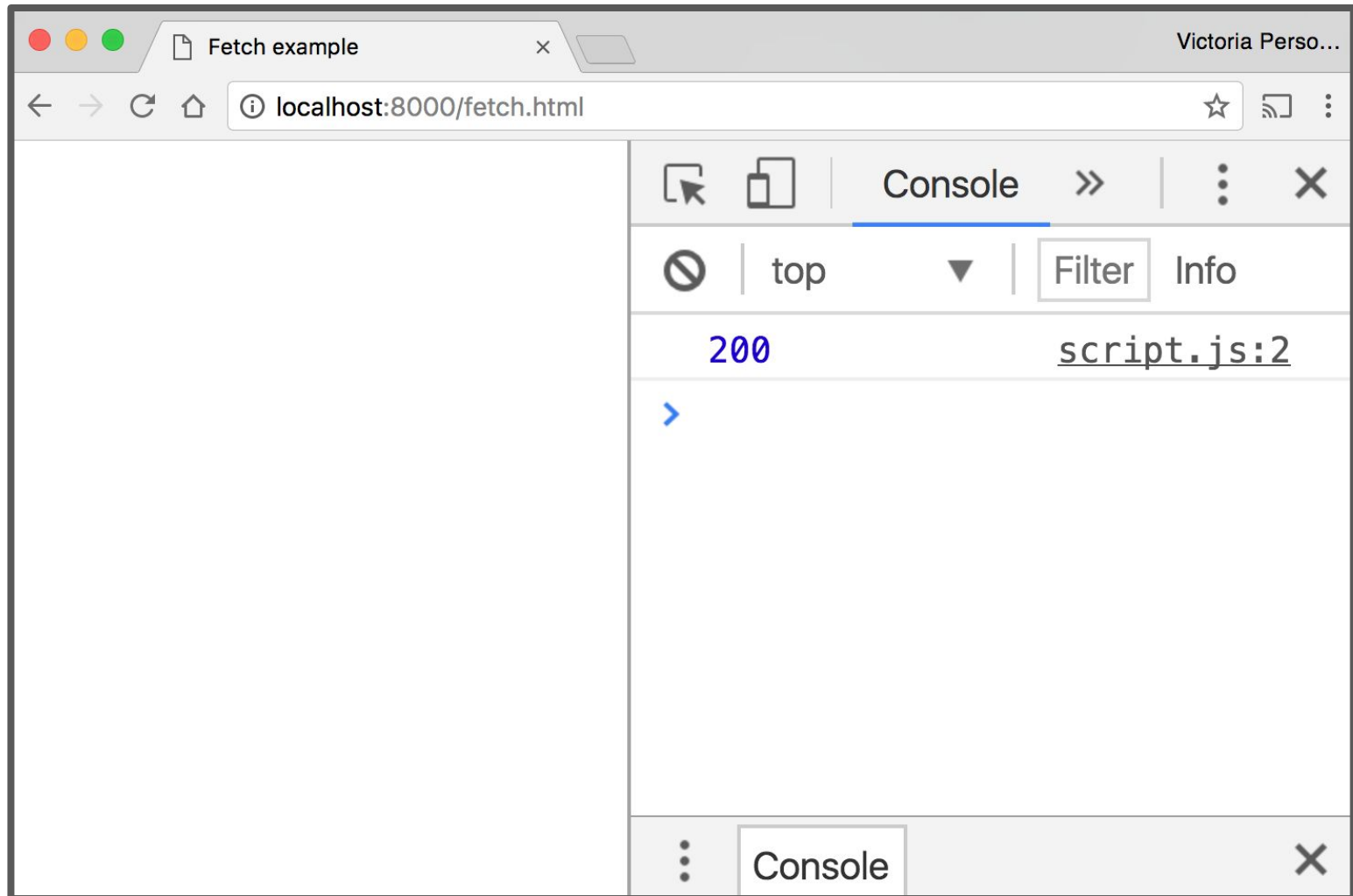
```
$ python -m SimpleHTTPServer  
Serving HTTP on 0.0.0.0 port 8000 ...
```

This now starts up a **server** that can load the files in the current directory over HTTP.

- We can access this server by navigating to:
<http://localhost:8000/>

```
$ python -m SimpleHTTPServer  
Serving HTTP on 0.0.0.0 port 8000 ...
```





We got HTTP response 200, which is success! ([codes](#))

How do we get the data from `fetch()`?

Using Fetch

```
function onSuccess(response) {  
    ..  
}  
fetch('images.txt').then(onSuccess);
```

- `response.status`: Status code for the request
- `response.text()`:
 - Asynchronously reads the response stream
 - **Returns a Promise** that resolves with the string containing the response stream data.

text() Promise

Q: How do we change the following code to print out the response body?

```
function onSuccess(response) {  
    console.log(response.status);  
}
```

```
function onError(error) {  
    console.log('Error: ' + error);  
}
```

```
fetch('images.txt')  
    .then(onSuccess, onError);
```

```
function onStreamProcessed(text) {  
    console.log(text);  
}
```

```
function onResponse(response) {  
    console.log(response.status);  
    response.text().then(onStreamProcessed);  
}
```

```
function onError(error) {  
    console.log('Error: ' + error);  
}
```

```
fetch('images.txt').then(onResponse, onError);
```

Chaining Promises

We want the following asynchronous actions to be completed in this order:

1. When the `fetch` completes, run `onResponse`
2. When `response.text()` completes, run `onStreamProcessed`

```
function onStreamProcessed(text) { ... }  
function onResponse(response) {  
  response.text().then(onStreamProcessed);  
}  
fetch('images.txt').then(onResponse, onError);
```

We can rewrite this:

```
function onStreamProcessed(text) {  
  console.log(text);  
}
```

```
function onResponse(response) {  
  response.text().then(onStreamProcessed);  
}
```

```
function onError(error) {  
  console.log('Error: ' + error);  
}
```

```
fetch('images.txt').then(onResponse, onError);
```

We can rewrite this:

```
function onStreamProcessed(text) {  
  console.log(text);  
}
```

```
function onResponse(response) {  
  return response.text();  
}
```

```
function onError(error) {  
  console.log('Error: ' + error);  
}
```

```
fetch('images.txt')  
  .then(onResponse, onError)  
  .then(onStreamProcessed);
```

Chaining Promises

```
function onStreamProcessed(text) {  
  console.log(text);  
}
```

```
function onResponse(response) {  
  return response.text();  
}
```

```
fetch('images.txt')  
  .then(onResponse, onError)  
  .then(onStreamProcessed);
```

Chaining Promises

```
function onStreamProcessed(text) {  
  console.log(text);  
}
```

```
function onResponse(response) {  
  return response.text();  
}
```

```
const responsePromise = fetch('images.txt')  
  .then(onResponse, onError)  
responsePromise.then(onStreamProcessed);
```

The Promise returned by `onResponse` is effectively* the Promise returned by `fetch`. (*Not actually what's happening, but that's how we'll think about it for right now.)

Chaining Promises

```
function onStreamProcessed(text) {  
  console.log(text);  
}
```

```
function onResponse(response) {  
  return response.text();  
}
```

```
fetch('images.txt')  
  .then(onResponse, onError)  
  .then(onStreamProcessed);
```

**If we don't think
about it too hard, the
syntax is fairly
intuitive.**

We'll think about this more
deeply later!

Completed example

```
function onStreamProcessed(text) {
  const urls = text.split('\n');
  for (const url of urls) {
    const image = document.createElement('img');
    image.src = url;
    document.body.append(image);
  }
}
function onSuccess(response) {
  response.text().then(onStreamProcessed)
}
function onError(error) {
  console.log('Error: ' + error);
}

fetch('images.txt').then(onSuccess, onError);
```

JSON

JavaScript Object Notation

JSON: Stands for **JavaScript Object Notation**

- Created by Douglas Crockford
- Defines a way of **serializing** JavaScript objects
 - **to serialize:** to turn an object into a string that can be deserialized
 - **to deserialize:** to turn a serialized string into an object
- `JSON.stringify(object)` returns a string representing *object* serialized in JSON format
- `JSON.parse(jsonString)` returns a JS object from the *jsonString* serialized in JSON format

JSON.stringify()

We can use the `JSON.stringify()` function to serialize a JavaScript object:

```
const bear = {  
  name: 'Ice Bear',  
  hobbies: ['knitting', 'cooking', 'dancing']  
};
```

```
const serializedBear = JSON.stringify(bear);  
console.log(serializedBear);
```

[CodePen](#)

JSON.parse()

We can use the `JSON.parse()` function to deserialize a JavaScript object:

```
const bearString = '{"name":"Ice  
Bear","hobbies":["knitting","cooking","danci  
ng"]}';
```

```
const bear = JSON.parse(bearString);  
console.log(bear);
```

[CodePen](#)

Why JSON?

JSON is a useful format for storing data that we can load into a JavaScript API via `fetch()`.

Let's say we had a list of Songs and Titles.

- If we stored it as a text file, we would have to know how we are separating song name vs title, etc
- If we stored it as a JSON file, we can just deserialize the object.

JSON

songs.json

```
1 {
2   "cranes": {
3     "fileName": "solange-cranes-kaytranada.mp3",
4     "artist": "Solange",
5     "title": "Cranes in the Sky [KAYTRANADA Remix]"
6   },
7   "timeless": {
8     "fileName": "james-blake-timeless.mp3",
9     "artist": "James Blake",
10    "title": "Timeless"
11  },
12  "knock": {
13    "fileName": "knockknock.mp4",
14    "artist": "Twice",
15    "title": "Knock Knock"
16  },
17  "deep": {
18    "fileName": "janet-jackson-go-deep.mp3",
19    "artist": "Janet Jackson",
20    "title": "Go Deep [Alesia Remix]"
21  },
22  "discretion": {
23    "fileName": "mitis-innocent-discretion.mp3",
24    "artist": "MitiS",
25    "title": "Innocent Discretion"
26  },
27  "spear": {
28    "fileName": "toby-fox-spear-of-justice.mp3",
29    "artist": "Toby Fox",
30    "title": "Spear of Justice"
31  }
32 }
```

Fetch API and JSON

The Fetch API also has built-in support for json:

```
function onStreamProcessed(json) {  
  console.log(json);  
}
```

```
function onResponse(response) {  
  return response.json();  
}
```

```
fetch('songs.json')  
  .then(onResponse, onError)  
  .then(onStreamProcessed);
```

JS and DOM

Context:

ECMAScript used to manipulate HTML (and not only)

Needs API for that

Initially browsers did their own things

Then W3C (World Wide Web Consortium) standards

Document Object Model

DOM1-3

Currently WHATWG «Living Standard» for HTML/DOM
(Living Standard is slightly oxymoronic)

Can just as well treat MDN as standard now...

JS and DOM

Context:

ECMAScript used to manipulate HTML (and not only)

Needs API for that

Initially browsers did their own things

Then W3C (World Wide Web Consortium) standards

Document Object Model

DOM1-3

Currently WHATWG «Living Standard» for HTML/DOM

(Living Standard is slightly oxymoronic)

Can just as well treat MDN as standard now...

JS and DOM

Context:

- ECMAScript used to manipulate HTML (and not only)

- Needs API for that

- Initially browsers did their own things

- Then W3C (World Wide Web Consortium) standards

- Document Object Model

- DOM1-3

- Currently WHATWG «Living Standard» for HTML/DOM

 - (Living Standard is slightly oxymoronic)

- Can just as well treat MDN as standard now...

JS and DOM — Basic functionality

Find: `document.querySelector(css_selector)`

Find many: `document.querySelectorAll(css_selector)`

Create: `document.createElement(tagname)`

Configure: `document.setAttribute(attrName, attrValue)`

Inspect: `document.getAttribute(attrName)`

Place: `element.appendChild(childElement)`

Place: `element.insertBefore(childElement, oldChildElement)`

Debug: `window.alert(message), console.log(object)`
... can inspect objects in browser console

JS and DOM — Basic functionality

Find: `document.querySelector(css_selector)`

Find many: `document.querySelectorAll(css_selector)`

Create: `document.createElement(tagname)`

Configure: `document.setAttribute(attrName, attrValue)`

Inspect: `document.getAttribute(attrName)`

Place: `element.appendChild(childElement)`

Place: `element.insertBefore(childElement, oldChildElement)`

Debug: `window.alert(message), console.log(object)`
... can inspect objects in browser console

JS and DOM — Basic functionality

Find: `document.querySelector(css_selector)`

Find many: `document.querySelectorAll(css_selector)`

Create: `document.createElement(tagname)`

Configure: `document.setAttribute(attrName, attrValue)`

Inspect: `document.getAttribute(attrName)`

Place: `element.appendChild(childElement)`

Place: `element.insertBefore(childElement, oldChildElement)`

Debug: `window.alert(message), console.log(object)`
... can inspect objects in browser console

JS and DOM — Basic functionality

Find: `document.querySelector(css_selector)`

Find many: `document.querySelectorAll(css_selector)`

Create: `document.createElement(tagname)`

Configure: `document.setAttribute(attrName, attrValue)`

Inspect: `document.getAttribute(attrName)`

Place: `element.appendChild(childElement)`

Place: `element.insertBefore(childElement, oldChildElement)`

Debug: `window.alert(message), console.log(object)`
... can inspect objects in browser console

DOM: attributes and values

```
inputElement = document.createElement("input")
```

```
inputElement.setAttribute("type", "text")
```

VS

```
inputElement.type = "text"
```

```
inputElement.setAttribute("value", "qwe")
```

VS

```
inputElement.value = "qwe"
```

DOM: attributes and values

```
inputElement = document.createElement("input")
```

```
inputElement.setAttribute("type", "text")
```

VS

```
inputElement.type = "text"
```

```
inputElement.setAttribute("value", "qwe")
```

VS

```
inputElement.value = "qwe"
```

DOM: attributes and values

```
inputElement = document.createElement("input")
```

```
inputElement.setAttribute("type", "text")
```

VS

```
inputElement.type = "text"
```

```
inputElement.setAttribute("value", "qwe")
```

VS

```
inputElement.value = "qwe"
```

DOM: attributes and values

```
inputElement = document.createElement("input")
```

```
inputElement.setAttribute("type", "text")
```

VS

```
inputElement.type = "text"
```

same

```
inputElement.setAttribute("value", "qwe")
```

VS

```
inputElement.value = "qwe"
```

DOM: attributes and values

```
inputElement = document.createElement("input")
```

```
inputElement.setAttribute("type", "text")
```

VS

```
inputElement.type = "text"
```

same

```
inputElement.setAttribute("value", "qwe")
```

VS

```
inputElement.value = "qwe"
```

initial vs. runtime-current

`element.style (structured)`

`element.classList`

DOM: Filling nodes

```
element.innerText = "qwe"
```

```
element.innerHTML = "<span>qwe</span>" (cheating a bit)
```

```
element.innerHTML = `<span>${3+4}</span>`
```

...beware of untrustworthy input!