6.170: Software Studio

JavaScript: The DOM and Event Handling

Arvind Satyanarayan +
Daniel Jackson
PREVIOUSLY IN 6.170...
Data Types

**Boolean**
`true, false`

**Numbers**
`0, 1, 3.14, 6.28`

**Strings**
"6170's the best"

**Arrays**
`[6170, true, "best"]`

**Objects**
`{"6170": "best"}`

**No Value**
`null, undefined`
Previously in 6.170...

Data Types
Loose Dynamic Typing
Previously in 6.170...

Data Types
Loose Dynamic Typing
Variables & Scope – \texttt{let}, \texttt{const}, \texttt{var}
Previously in 6.170...

Data Types
Loose Dynamic Typing
Variables & Scope—let, const, var
First-Class Functions: Functions as Values

```javascript
let multiply = function(a, b) {
    return a * b;
}

let toys = [
    {name: "Woody", price: 10},
    {name: "Buzz", price: 15},
    ...
];

toys = toys.filter(function(toy) {
    return toy.price < 9;
});
```
Data Types
Loose Dynamic Typing
Variables & Scope—`let`, `const`, `var`
First-Class Functions:
  Functions as Values
List Functionals

```javascript
// Previously in 6.170...

let toys = toys.filter(function(toy) {
  return toy.price < 9;
});

let toys = toys.map(function(toy) {
  toy.price = toy.price * 2;
  return toy;
});

let toys =
  toys.reduce(function(sum, toy) {
    return sum + toy.price;
  }, 0);
```
Previously in 6.170...

Data Types
Loose Dynamic Typing
Variables & Scope—let, const, var
First-Class Functions:
  Functions as Values
List Functionals
Closures

```javascript
let suffix = "!";
function greet(prefix) {
  return function(msg) {
    console.log(prefix + msg + suffix);
  }
}

let hello = greet("hello ");
hello("arvind"); -> "hello arvind!"

suffix = "?";
hello("daniel"); -> "hello daniel?"
```
Previously in 6.170...

Data Types
Loose Dynamic Typing
Variables & Scope—let, const, var
First-Class Functions:
  Functions as Values
List Functionals
Closures
Mixins

```javascript
let person = {
  name: "arvind",
  greet(punc) {
    return this.name + punc;
  },
  ...
};

let animal = {
  name: "panda"
};

person.greet.call(animal, "!");
-> "panda!"
```
Today

Functions as Constructors

Prototypal Inheritance

DOM Manipulation

Event Handling
Functions as *constructors*

Functions can also be used to make custom objects, similar to traditional class definitions.
Functions as *constructors*

Functions can also be used to make custom objects, similar to traditional class definitions.

```javascript
function Color(r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b;
}
```
Functions as constructors

Functions can also be used to make custom objects, similar to traditional class definitions.

The `new` operator creates a new object and is passed as the context of the constructor.

```javascript
function Color(r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b;
}

let c = new Color(255, 0, 0);
-> Color {r: 255, g: 0, b: 0}
```
Functions as constructors

Functions can also be used to make custom objects, similar to traditional class definitions.

The `new` operator creates a new object and is passed as the context of the constructor.

```javascript
function Color(r, g, b) {
  this.r = r;
  this.g = g;
  this.b = b;
}

let c = new Color(255, 0, 0);
-> Color {r: 255, g: 0, b: 0}

c instanceof Color
-> true
Functions as constructors

Functions can also be used to make custom objects, similar to traditional class definitions.

The `new` operator creates a new object and is passed as the context of the constructor.

Final step for object-oriented programming is inheritance.

```javascript
function Color(r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b;
}

let c = new Color(255, 0, 0);
-> Color {r: 255, g: 0, b: 0}
c instanceof Color
-> true
```
Inheritance via object *prototypes*

All objects have a *prototype* property.
Inheritance via object *prototypes*

All objects have a *prototype* property.

```javascript
b = {bits: 24};
console.log(b.__proto__);
-> {toString: f, ...}
```
Inheritance via object *prototypes*

All objects have a *prototype* property.

```javascript
b = {bits: 24};
color = Object.create(b);
```
Inheritance via object *prototypes*

All objects have a *prototype* property.

Prototypes form a chain.

```javascript
b = {bits: 24};
color = Object.create(b);
color.__proto__ === b;
-> true
```
Inheritance via object prototypes

All objects have a prototype property.

Prototypes form a chain, and property/method lookups propagates down the chain.

```
b = {bits: 24};
color = Object.create(b);
color.r = 255;
color.b = 0;
color.g = 0;
console.log(color.r);
```
Inheritance via object *prototypes*

All objects have a *prototype* property.

Prototypes form a chain, and property/method lookups propagates down the chain.

```
b = {bits: 24};
color = Object.create(b);
color.r = 255;
color.b = 0;
color.g = 0;
console.log(color.r);
-> 255
```
Inheritance via object *prototypes*

All objects have a *prototype* property.

Prototypes form a chain, and property/method lookups propagates down the chain.

```javascript
b = {bits: 24};
color = Object.create(b);
color.r = 255;
color.b = 0;
color.g = 0;
console.log(color.bits);
```
Inheritance via object prototypes

All objects have a prototype property.

Prototypes form a chain, and property/method lookups propagates down the chain.

```
b = {bits: 24};
color = Object.create(b);
color.r = 255;
color.b = 0;
color.g = 0;
console.log(color.bits);
```
Inheritance via object *prototypes*

All objects have a *prototype* property.

Prototypes form a chain, and property/method lookups propagates down the chain.

```javascript
b = {bits: 24};
color = Object.create(b);
color.r = 255;
color.b = 0;
color.g = 0;
console.log(color.bits);
-> 24
```
All objects have a *prototype* property.

Prototypes form a chain, and property/method lookups propagates down the chain.

```javascript
b = {bits: 24};
color = Object.create(b);

color.r = 255;
color.b = 0;
color.g = 0;

console.log(color.bits);

b.bits = 48;
console.log(color.bits);
```
Inheritance via object *prototypes*

All objects have a *prototype* property.

Prototypes form a chain, and property/method lookups propagates down the chain.

```javascript
b = {bits: 24};
color = Object.create(b);

color.r = 255;
color.b = 0;
color.g = 0;

console.log(color.bits);
-> 24

b.bits = 48;
console.log(color.bits);
```
Inheritance via object prototypes

All objects have a prototype property.

Prototypes form a chain, and property/method lookups propagates down the chain.

```
b = {bits: 24};
color = Object.create(b);
color.r = 255;
color.b = 0;
color.g = 0;
console.log(color.bits);
```

```
b.bits = 48;
console.log(color.bits);
```

`null`
Inheritance via object *prototypes*

All objects have a *prototype* property.

Prototypes form a chain, and property/method lookups propagates down the chain.

```
b = {bits: 24};
color = Object.create(b);
color.r = 255;
color.b = 0;
color.g = 0;
```

```
console.log(color.bits);
```

```
b.bits = 48;
color.bits = 16;
```

```
console.log(color.bits);
```
Inheritance via object prototypes

All objects have a prototype property.

Prototypes form a chain, and property/method lookups propagates down the chain.

```javascript
b = {bits: 24};
color = Object.create(b);

color.r = 255;
color.b = 0;
color.g = 0;

console.log(color.bits);
```

```javascript
b.bits = 48;
color.bits = 16;
console.log(color.bits);
-> 16
```
Inheritance via object prototypes

All objects have a prototype.

Prototypes form a chain, and property/method lookups propagates down the chain.

```javascript
function Color(r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b;
}
```
Inheritance via object *prototypes*

All objects have a *prototype*.

Prototypes form a chain, and property/method lookups propagates down the chain.

```javascript
function Color(r, g, b) {
  this.r = r;
  this.g = g;
  this.b = b;
}

Color.prototype = Object.create({bits: 24});
```
Inheritance via object prototypes

All objects have a prototype.

Prototypes form a chain, and property/method lookups propagates down the chain.

```javascript
function Color(r, g, b) {
  this.r = r;
  this.g = g;
  this.b = b;
  Bits.call(this);
}

Color.prototype = Object.create(Bits.prototype);
```
Inheritance via object prototypes

All objects have a prototype.

Prototypes form a chain, and property/method lookups propagates down the chain.

```javascript
function Color(r, g, b) {
  this.r = r;
  this.g = g;
  this.b = b;
  Bits.call(this);
}

Color.prototype = Object.create(Bits.prototype);

Color.prototype.hex = function() {
  return "#" + ...
  this.r.toString(16) + ...
  this.g.toString(16) + ...
  this.b.toString(16);
}
```
Inheritance via object prototypes

All objects have a prototype.

Prototypes form a chain, and property/method lookups propagates down the chain.

```javascript
function Color(r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b;
    Bits.call(this);
}

Color.prototype = Object.create(Bits.prototype);

Color.prototype.hex = function() {
    return "#" + ... 
    this.r.toString(16) + ... 
    this.g.toString(16) + ... 
    this.b.toString(16);
}

let c = new Color(255, 0, 0); 
console.log(c.hex());
```
ES6 **classes** provide syntactic sugar

```javascript
function Color(r, g, b) {
  this.r = r;
  this.g = g;
  this.b = b;
  Bits.call(this);
}

Color.prototype = Object.create(Bits.prototype);
Color.prototype.hex = function() {
  return "#" + ...
    this.r.toString(16) + ...
    this.g.toString(16) + ...
    this.b.toString(16);
}

let c = new Color(255, 0, 0);
console.log(c.hex());
```

```javascript
class Color extends Bits {
  constructor(r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b;
    super();
  }

  hex = function() {
    return "#" + ...
      this.r.toString(16) + ...
      this.g.toString(16) + ...
      this.b.toString(16);
  }
}

let c = new Color(255, 0, 0);
console.log(c.hex());
```
Prototypes vs. Classical Inheritance

- **Dynamic Inheritance:** Properties can be added to prototypes on the fly.
- **Prototype chain can be modified on the fly.**
- **More Expressive:** Rather than a single prototype chain, can do lookup propagation across more than one prototype for multiple inheritance.

In practice, prototypes are used to just model classical inheritance.

```
class Color extends Bits {
  constructor(r, g, b) {
    this.r = r;
    this.g = g;
    this.b = b;
    super();
  }
  hex = function() {
    return "#" + ...
    this.r.toString(16) + ...
    this.g.toString(16) + ...
    this.b.toString(16);
  }
}
let c = new Color(255, 0, 0);
console.log(c.hex());
```
DOM Manipulation
<!DOCTYPE html>
<html>
<head>
  <title>My First Webpage!</title>
</head>
<body>
  <h1 id="header">Hello
      <a href="http://mit.edu">World</a>
  </h1>

  <p class="caption">
    Thunder, thunder, thundercats, Ho!
  </p>

  ...
</body>
</html>
<!DOCTYPE html>
<html>
  <head>
    <title>My First Webpage!</title>
  </head>
  <body>
    <h1 id="header">Hello
      <a href="http://mit.edu">World</a>
    </h1>
    
    <p class="caption">
      Thunder, thunder, thundercats, Ho!
    </p>
    ...
  </body>
</html>
<!DOCTYPE html>
<html>
  <head>
    <title>My First Webpage!</title>
  </head>
  <body>
    <h1 id="header">Hello
        <a href="http://mit.edu">World</a>
    </h1>

    <p class="caption">
        Thunder, thunder, thundercats, Ho!
    </p>

    ... 
  </body>
</html>
<!DOCTYPE html>
<html>
 <head>
  <title>My First Webpage!</title>
 </head>
 <body>
  <h1 id="header">Hello
   <a href="http://mit.edu">World</a>
  </h1>
  
  <p class="caption">
   Thunder, thunder, thundercats, Ho!
  </p>
  ...
 </body>
</html>
<!DOCTYPE html>
<html>
<head>
  <title>My First Webpage!</title>
</head>
<body>
  <h1 id="header">Hello
      <a href="http://mit.edu">World</a>
    </h1>

  <p class="caption">
    Thunder, thunder, thundercats, Ho!
  </p>

  ...
</body>
</html>
<!DOCTYPE html>
<html>
  <head>
    <title>My First Webpage!</title>
  </head>
  <body>
    <h1 id="header">Hello
      <a href="http://mit.edu">World</a>
    </h1>
    
    <p class="caption">
      Thunder, thunder, thundercats, Ho!
    </p>
    
    ...  
  </body>
</html>
Selecting a DOM Node

```
<html>
  <head>
    <title>My first webpage</title>
  </head>
  <body>
    <a>Hello</a>
    <h1>Hello World</h1>
  </body>
</html>
```
**Selecting a DOM Node**

Historically:

- `document.getElementById("header")`
- `document.getElementsByClassName("caption")`
- `document.getElementsByTagName("h1")`
Selecting a DOM Node

Historically:

```javascript
document.getElementById("header")
document.getElementsByClassName("caption")
document.getElementsByTagName("h1")
```

With Selectors:

```javascript
document.querySelector("#header")
document.querySelectorAll(".caption")
```
Selecting a DOM Node

Historically:

```javascript
document.getElementById("header")
document.getElementsByClassName("caption")
document.getElementsByTagName("h1")
```

With Selectors:

```javascript
document.querySelector("#header")
document.querySelectorAll(".caption")
```

(except pseudo-classes e.g., .caption:hover)
DOM Nodes vs. Elements

**Element**: DOM Node that corresponds to an HTML element e.g., `<h1>` or `<p>`.
DOM Nodes vs. Elements

**Element**: DOM Node that corresponds to an HTML element e.g., `<h1>` or `<p>`.

**Node**: Generic DOM Node class. Other node types include Document.
**DOM Nodes vs. Elements**

**Element**: DOM Node that corresponds to an HTML element e.g., `<h1>` or `<p>`.

**Node**: Generic DOM Node class. Other node types include Document, Text,
**DOM Nodes vs. Elements**

**Element**: DOM Node that corresponds to an HTML element e.g., `<h1>` or `<p>`.

**Node**: Generic DOM Node class. Other node types include `Document`, `Text`, `Comment`, ...
DOM Nodes vs. Elements

Element: DOM Node that corresponds to an HTML element e.g., `<h1>` or `<p>`.

Node: Generic DOM Node class. Other node types include Document, Text, Comment, ...

element.tagName
element.id
element.className
element.innerHTML
element.style
**DOM Nodes vs. Elements**

**Element**: DOM Node that corresponds to an HTML element e.g., `<h1>` or `<p>`.

**Node**: Generic DOM Node class. Other node types include `Document`, `Text`, `Comment`, ...

- `element.tagName`
- `element.id`
- `element.className`
- `element.innerHTML`
- `element.style`

- `node.nodeType`
- `node.nodeValue`
Manipulating Properties

- document
  - html
    - head
      - title
        - "My first webpage"
    - body
      - h1
        - "Hello"
      - a
        - "World"
Manipulating Properties

```javascript
element.style.color = "steelblue";
element.style.borderRadius = "10px";
```
Manipulating Properties

element.style.color = "steelblue";
element.style.borderRadius = "10px";
camelCase for multiword properties.
Manipulating Properties

element.style.color = "steelblue";
element.style.borderRadius = "10px";

element.innerHTML = "<p>Pork belly mixtape</p>";
Manipulating Properties

element.style.color = "steelblue";
element.style.borderRadius = "10px";

element.innerHTML = "<p>Pork belly mixtape</p>";

✘ String concatenation: difficult to maintain, easy to make mistakes. Poor performance as browser has to first parse the string, then reconstruct DOM subtree.

✘ Impossible to make fine-grained adjustments: e.g., insert element between two others.

✘ Doesn't return references to new elements (e.g., for further manipulation).
Manipulating the Tree

```
document
  html
    head
    body
      ... h1
        "Hello"
        a
          "World"
```
Manipulating the Tree

**Goal:** Add another link (a) after the h1.
Manipulating the Tree

Goal: Add another link (a) after the h1.

let link = document.createElement("a");
Manipulating the Tree

**Goal:** Add another link (a) after the h1.

```javascript
let link = document.createElement("a");
link.href = "http://eecs.mit.edu";
```
Manipulating the Tree

**Goal:** Add another link (a) after the h1.

```javascript
let link = document.createElement("a");
link.href = "http://eecs.mit.edu";

let label = document.createTextNode("EECS");
```

Manipulating the Tree

**Goal:** Add another link (a) *after* the h1.

```javascript
let link = document.createElement("a");
link.href = "http://eecs.mit.edu";

let label = document.createTextNode("EECS");
link.appendChild(label);
```
**Goal:** Add another link (a) after the h1.

```javascript
let link = document.createElement("a");
link.href = "http://eecs.mit.edu";

let label = document.createTextNode("EECS");

link.appendChild(label);

let body = document.querySelector("body");
body.appendChild(link);
```
Manipulating the Tree

**Goal:** Add another link (a) *after* the h1.

```javascript
let link = document.createElement("a");
link.href = "http://eecs.mit.edu";

let label = document.createTextNode("EECS");
link.appendChild(label);

let body = document.querySelector("body");
body.appendChild(link);
```
Manipulating the Tree

```
<html>
  <head>
    ...
  </head>
  <body>
    <h1>Hello</h1>
    <a>World</a>
  </body>
</html>
```
Manipulating the Tree

```javascript
let newChild = document.createElement("p");
```
Manipulating the Tree

```javascript
let newChild = document.createElement("p");

let parent = document.querySelector("body");
let sibling = document.querySelector("h1");
parent.insertBefore(newChild, sibling);
```
Manipulating the Tree

```javascript
let newChild = document.createElement("p");
let parent = document.querySelector("body");
let sibling = document.querySelector("h1");
parent.insertBefore(newChild, sibling);
parent.replaceChild(newChild, sibling);
parent.removeChild(newChild, sibling);
```
Traversing the Tree

```javascript
let parent = document.querySelector(...);

parent.childNodes / parent.children
parent.parentNode / parent.parentElement
parent.previousSibling / parent.previousElementSibling
parent.nextSibling / parent.nextElementSibling
```

Diagram:
- `document`
- `html`
- `head`
- `body`
- `h1`
- `a`
- "Hello"
- "World"
- "EECS"
In-Class Activity! Submit To: yellkey.com/toward

In a web browser, go to http://example.com and open your JavaScript Console.

1. Write three ways of selecting the "More information" link, one of which must include a DOM tree traversal starting from the body tag.

2. Inside the white box, add a third paragraph of text.

3. Add a 2nd-level heading before the paragraph you just added.

We recommend you copy + paste your code into the form after each part. That way, you can refresh example.com if you make a mistake, and start fresh.
In-Class Activity! (Solutions)

1. Write three ways of selecting the "More information" link, one of which must include a DOM tree traversal starting from the body tag.
In-Class Activity! (Solutions)

2. Inside the white box, add a third paragraph of text.
3. Add a 2nd-level heading before the paragraph you just added.
Events
JavaScript is *single threaded*. 
JavaScript is single threaded. It can do only one thing at a time.
JavaScript is single threaded. It can do only one thing at a time. Long-running operations (e.g., animation, network requests) would block the user interface.
The Event Queue & Loop
The Event Queue & Loop

JavaScript has a traditional **call stack** that tracks function execution.
JavaScript has a traditional *call stack* that tracks function execution.

It also has an *event queue* (or *message queue*).
The **Event Queue & Loop**

JavaScript has a traditional *call stack* that tracks function execution.

It also has an *event queue* (or *message queue*).

Events can include:

- **Interaction**: clicks, hovers, mouse move, keyboard presses, etc.
- **Timer/Animation**
- **Resources & Network Requests**: load/unload
The **Event Queue & Loop**

When the call stack is empty, the *event loop* evaluates:
The **Event Queue & Loop**

When the call stack is empty, the *event loop* evaluates:

```javascript
while (queue.waitForMessage()) {
    queue.processNextMessage();
}
```

![Diagram of call stack and message queue](image-url)
The **Event Queue & Loop**

When the call stack is empty, the *event loop* evaluates:

```javascript
while (queue.waitForMessage()) {
    queue.processNextMessage();
}
```

Events are processed in the order they were added to the queue, and run to completion.
The Event Queue & Loop: An Example

```javascript
function one() {
    console.log("one");
    setTimeout(two, 0);
    console.log("three");
    setTimeout(four, 0);
    console.log("five");
}

function two() {
    console.log("two");
}

function four() {
    console.log("four");
}
```
The Event Queue & Loop: An Example

function one() {
    console.log("one");
    setTimeout(two, 0);
    console.log("three");
    setTimeout(four, 0);
    console.log("five");
}

function two() {
    console.log("two");
}

function four() {
    console.log("four");
}

Adds a *timer event* to the queue to execute the function after a (minimum) specified delay.
The Event Queue & Loop: An Example

function one() {
  console.log("one");
  setTimeout(two, 0);
  console.log("three");
  setTimeout(four, 0);
  console.log("five");
}

function two() {
  console.log("two");
}

function four() {
  console.log("four");
}

Adds a timer event to the queue to execute the function after a (minimum) specified delay.

What gets printed to the console when we call `one()`?
```javascript
function one() {
    console.log("one");
    setTimeout(two, 0);
    console.log("three");
    setTimeout(four, 0);
    console.log("five");
}

function two() {
    console.log("two");
}

function four() {
    console.log("four");
}
```

```bash
> one()
  one
  three
  five
  undefined
> undefined
  two
  four
```
1. The event queue only evaluates once the call stack is empty (i.e., everything has finished evaluating).
1. The event queue only evaluates once the call stack is empty (i.e., everything has finished evaluating).

2. Events execute in the order they were added to the queue, and run to completion.
Adding **Timer** Events

setTimeout(function() {...}, ms);
Executes the given function after a minimum delay (specified in milliseconds).

let timerID = setInterval(function() {...}, ms);
Executes the given function at a given time interval (in milliseconds). The return value can be passed to clearInterval(timerID), to stop the timer.
Listening for Interaction Events

let elem = document.querySelector("h1");
elem.addEventListener("click", function(event) {...});
Let `elem = document.querySelector("h1");`
`elem.addEventListener("click", function(event) {...});`;

Called a *listener* (or event *handler*) because the browser automatically creates events + adds to the queue when the heading is clicked.
Listening for Interaction Events

```javascript
let elem = document.querySelector("h1");
elem.addEventListener("click", function(event) {...});
```

Called a *listener* (or event *handler*) because the browser automatically creates events + adds to the queue when the heading is clicked.

We *listen* for the events: when the event is processed from the queue, the registered listener function is called.
Listening for Interaction Events

```
let elem = document.querySelector("h1");
elem.addEventListener("click", function(event) {...});
```

Called a listener (or event handler) because the browser automatically creates events + adds to the queue when the heading is clicked.

We listen for the events: when the event is processed from the queue, the registered listener function is called.

The function receives an event argument, and the function context (this) identifies which DOM element the listener is being called for.
In-Class Activity!  
Submit To: yellkey.com/toward

In a web browser, go to http://example.com and open your JavaScript Console.

1. Change the font color of the h1 heading every time you click it.

2. Change the background color of the center box every second.

Be creative!


We recommend you copy + paste your code into the form after each part. That way, you can refresh example.com if you make a mistake, and start fresh.
In-Class Activity!

1. Change the font color of the h1 heading every time you click it.
In-Class Activity!

2. Change the background color of the center box every second.
Single-Threaded + Running to Completion

Example adapted from Eirik Bakke.
Event Handling Gotchas: *Closures*

```javascript
for (i = 0; i < 3; i++) {
    let btn = document.createElement("button");
    btn.innerText = i;

    btn.addEventListener("click", function() {
        console.log(i);
    });

    document.body.append(btn);
}
```
Event Handling Gotchas: *Closures*

```javascript
for (i = 0; i < 3; i++) {
  let btn = document.createElement("button");
  btn.innerText = i;

  btn.addEventListener("click", function() {
    console.log(i);
  });

  document.body.append(btn);
}
```

**Q.** What will be logged when you click button 1 vs. button 2 vs. button 3?
for (i = 0; i < 3; i++) {
    let btn = document.createElement("button");
    btn.innerText = i;

    btn.addEventListener("click", function() {
        console.log(i);
    });

    document.body.append(btn);
}

Q. What will be logged when you click button 1 vs. button 2 vs. button 3?

A. The event handler forms a closure, which references i, a global variable.
Event Handling Gotchas: **Closures**

```javascript
for (let i = 0; i < 3; i++) {
    let btn = document.createElement("button");
    btn.innerText = i;

    btn.addEventListener("click", function() {
        console.log(i);
    });

    document.body.appendChild(i);
}
```

Use ES6's `let` binding, or ...
Event Handling Gotchas: **Context**

```javascript
function Counter() {
    this.count = 0;
    this.btn = document.createElement("button");
    this.btn.innerText = "Increment Me";

    this.btn.addEventListener("click", function() {
        this.count += 1;
        console.log(this.count);
    });

    document.body.appendChild(this.btn);
}

let count = new Counter();
```
Event Handling Gotchas: *Context*

```javascript
function Counter() {
  this.count = 0;
  this.btn = document.createElement("button");
  this.btn.innerText = "Increment Me";

  this.btn.addEventListener("click", function() {
    this.count += 1;
    console.log(this.count);
  });

  document.body.append(this.btn);
}

let count = new Counter();
```

Q. What will be logged when you click the button?
Event Handling Gotchas: **Context**

```javascript
function Counter() {
    this.count = 0;
    this.btn = document.createElement("button");
    this.btn.innerText = "Increment Me";

    this.btn.addEventListener("click", function() {
        this.count += 1;
        console.log(this.count);
    });

    document.body.append(this.btn);
}

let count = new Counter();
```

Q. What will be logged when you click the button?
Event Handling Gotchas: *Context*

```javascript
function Counter() {
    this.count = 0;
    this.btn = document.createElement("button");
    this.btn.innerText = "Increment Me";

    this.btn.addEventListener("click", function() {
        this.count += 1;
        console.log(this.count);
    });

    document.body.appendChild(this.btn);
}

let count = new Counter();
```

**Q.** What will be logged when you click the button?

**A.** Inside event handlers, `this` refers to the DOM element handling the event.
Event Handling Gotchas: **Context**

```javascript
function Counter() {
    this.count = 0;
    this.btn = document.createElement("button");
    this.btn.innerText = "Increment Me";

    this.btn.addEventListener("click", () => {
        this.count += 1;
        console.log(this.count);
    });

    document.body.append(this.btn);
}

let count = new Counter();
```

ES6 introduced "arrow functions", which *don't* have their own separate context.

So *this* now refers to the same object throughout.
**Event Handling Gotchas: ** Context

```javascript
function Counter() {
    this.count = 0;
    this.btn = document.createElement("button");
    this.btn.innerText = "Increment Me";
    var that = this;
    this.btn.addEventListener("click", function() {
        that.count += 1;
        console.log(that.count);
    });
    document.body.append(this.btn);
}

let count = new Counter();
```

Prior to ES6, need to create a separate variable, typically named "that," and handler would reference it via closure scope.