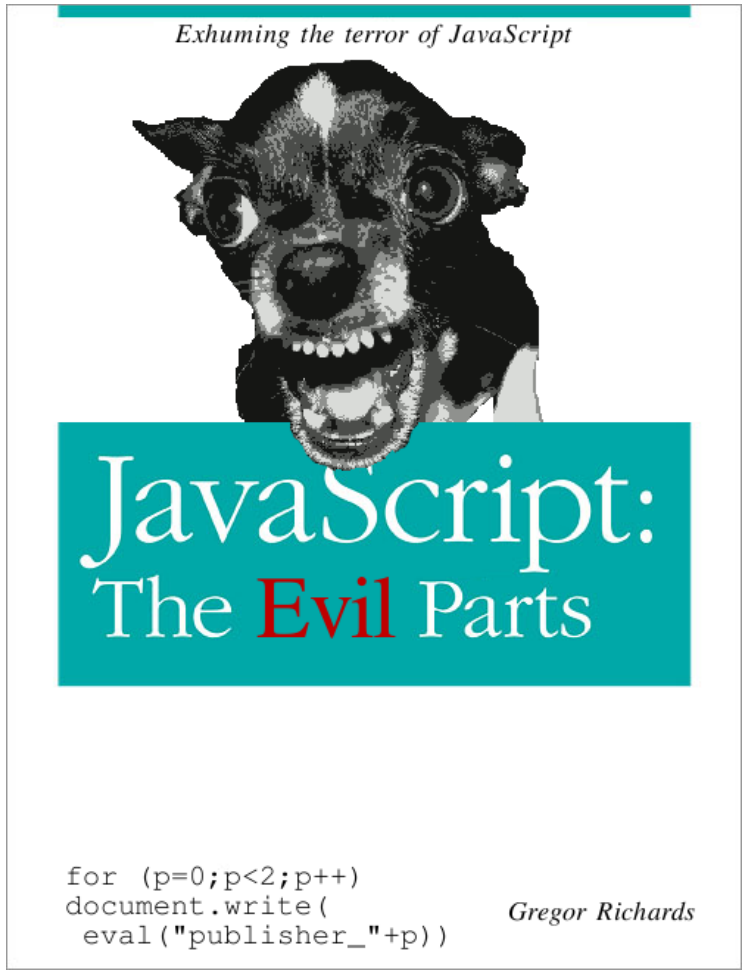
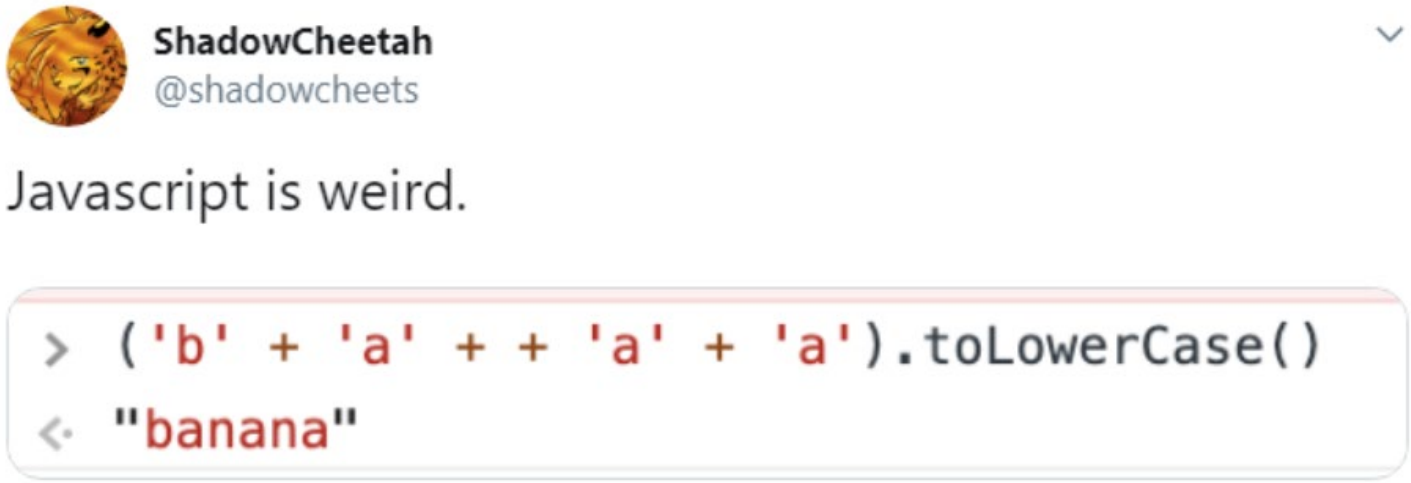
JavaScript Part I





Outline

- **Review: safety and idempotency**
- Intro to JavaScript
- Prototypes
- Event-Driven Programming

Review: REST APIs

- **REST (Representational State Transfer) APIs** expose functionality over the web through URLs
  - **API Endpoints** are URLs that represent some remote resource
    - Like a comment or a user profile
  - RESTful APIs generally use JSON (JavaScript Object Notation) to encode state that gets transferred between client and server
- REST APIs are used everywhere
  - Github, Spotify, YouTube, UMich, Coinbase, Mangadex, Gamesdonequick
    - Many more internally-used APIs
- REST is *not* a protocol, but a set of “best practices”

Review: safety and idempotency

- Safe
  - read-only
  - does not change server state
  - "What is your name?"

- Idempotent
  - Sending message multiple times has same effect on server state as sending once
  - Can safely retry
  - "Turn to page 394"

Thought question

- Which of these human requests are **safe**? Which are **idempotent**?

- “Can I ask you … your name?”
- "Can I have a bowl of rice?"
- "Turn to page three hundred, ninety-four."
- "I don't need your help anymore."
- "How many books are in the library?“
- “Say ‘what’ again.”

Idempotent

- Multiple identical requests should have the same effect *on the server* as a single request
- The same request can be made twice with no negative consequences *on the server*
- Does **not** mean that the same request always returns the same response
- **Does** mean that a request has NO side effects

- Why does Idempotency matter? If a request fails, can we automatically try again? Only if it is idempotent.

- Reference: HTTP RFC https://tools.ietf.org/html/rfc7231#section-4.2.2

Examples

- We'll use a small accounts API for the following examples

```
GET /accounts/1/ HTTP/1.0
HTTP/1.0 200 OK
{
  "name": "Tim Berners-Lee",
  "locked": False,
  "url": "/accounts/1/"
}
```

Not idempotent: POST

- POST is not idempotent
- POST creates a new object
- Call POST several times creates several new objects

```
POST /accounts/ HTTP/1.0
{
  "name": "Tim Berners-Lee",
  "locked": False,
}
HTTP/1.0 201 CREATED
{
  "name": "Tim Berners-Lee",
  "locked": False,
  "url": "/accounts/1/"
}
```

```
POST /accounts/ HTTP/1.0
{
  "name": "Tim Berners-Lee",
  "locked": False,
}
HTTP/1.0 201 CREATED
{
  "name": "Tim Berners-Lee",
  "locked": False,
  "url": "/accounts/2/"
}
```

Idempotent: DELETE

- DELETE removes the entire object
- Call DELETE twice, you get the same result *on the server*
  - Object is gone
- DELETE is idempotent

```
DELETE /accounts/1/ HTTP/1.0
HTTP/1.0 204 NO CONTENT

DELETE /accounts/1/ HTTP/1.0
HTTP/1.0 404 NOT FOUND
```

Idempotent: PUT

- PUT replaces the entire object
- Call PUT twice, you get the same result on the server
- PUT is idempotent

```
PUT /accounts/1/ HTTP/1.0

{
  "name": "Timmy Berners-Lee",
  "locked": False,
}

HTTP/1.0 200 OK
{
  "name": "Timmy Berners-Lee",
  "locked": False,
  "url": "/accounts/1/"
}
```

```
PUT /accounts/1/ HTTP/1.0

{
  "name": "Timmy Berners-Lee",
  "locked": False,
}

HTTP/1.0 200 OK
{
  "name": "Timmy Berners-Lee",
  "locked": False,
  "url": "/accounts/1/"
}
```

Review: REST APIs

- REST APIs use HTTP as a communication protocol
  - “verbs” consist of HTTP request methods
  - GET POST DELETE PATCH PUT

- GET is **safe** and **idempotent**
- POST is **unsafe** and **not idempotent**
- DELETE is **unsafe** but **idempotent**

- Curl is invaluable for checking results of API calls!

One Slide Summary: JavaScript (Part 1)

- JavaScript is a **dynamically typed, multi-paradigm** programming language
- JavaScript is a **weird language**
  - Functional language with imperative and object-oriented features
- We use JS to add dynamic behavior on the client in Web Services
- JavaScript is based around events and asynchronous behavior
  - onClick, setTimeout, onSuccess, etc.
  - Lots of anonymous functions…
    - onclick = function() { function cb1() { alert('hello'); }; setTimeout(cb1, 1000); }
- JavaScript **Objects** are created with Constructor functions
  - Objects are like **maps** from **property name** to **value** (i.e., field name or method)
  - JS allows modification of objects via **Prototypes**

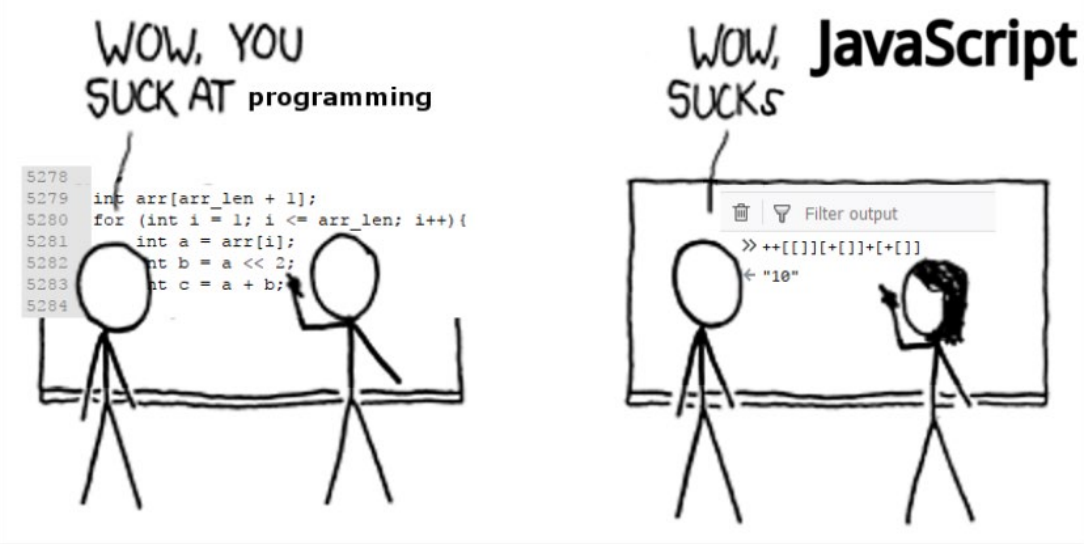
Outline

- Review: safety and idempotency
- **Intro to JavaScript**
- Prototypes
- Event-Driven Programming

JavaScript

- "Python with C++ syntax"
- With more design flaws

- “Only” programming language that web browsers support for client-side dynamic pages
  - Basically, we’re stuck with it



Review: server-side dynamic content

- What can't we do with server-side dynamic pages?

- Examples from P2:
  - Add comment without a page reload
  - Add like without a page reload
  - Delete without a page reload
  - Infinite scroll

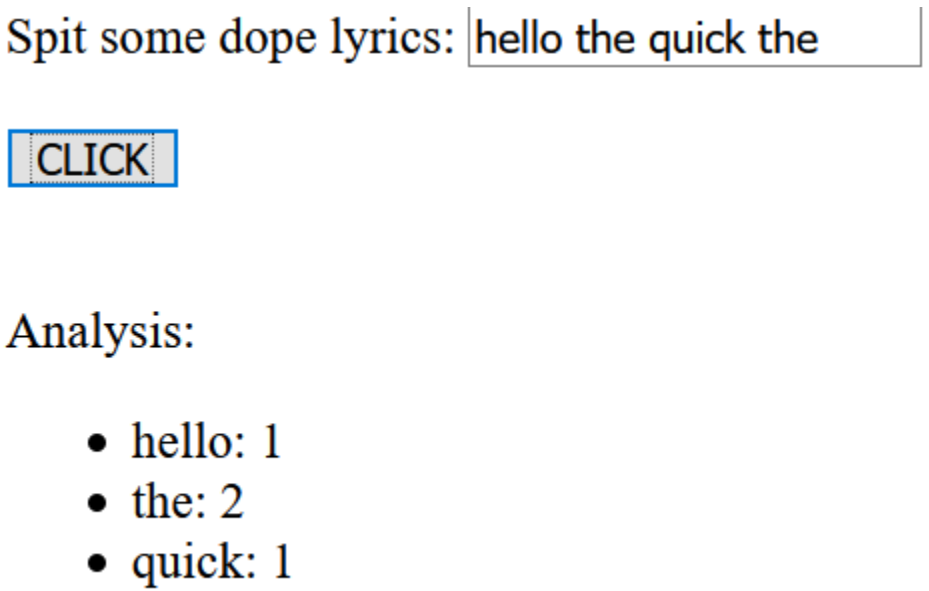
Other uses of JavaScript

JavaScript Example in HTML

```
<html>
<body>
  <p onClick="alert('Hello');">Click me</p>
  <p onMouseOver="x = x + 1; alert('You have moused over ' + x + ' times!');">MouseOver me</p>
</body>
</html>
```

More Dank JS Examples

- Could use make a client-side dynamic page that counts words in a user input?



JavaScript Concept: References

- Operations in JavaScript work with **references to objects** in memory or **values of primitives** in memory
- A JavaScript reference is like a C/C++ pointer
- Assignment of objects means **copying** the **pointer**

```
> let course = { name: 'Web Systems', num: 485 };
undefined
> eecs485 = course;
{ name: 'Web Systems', num: 485 }
> eecs485.name = 'Web';
'Web'
> eecs485
{ name: 'Web', num: 485 }
> course
{ name: 'Web', num: 485 }
```

JavaScript Concept: Object allocation

- Objects are allocated automatically on assignment in a heap

```
> let n = 123;        // allocates memory for a number
> let s = 'eecs485'; // allocates memory for a string
> let o = {a: 1, b: null}; // allocates memory for
                           // object, contained values
```

- Objects are deallocated automatically
  - Garbage collection
  - Combination of reference counting and cycle detection

- Objects are **prototype-based** (more slides later)

JavaScript concept: functions

- JavaScript functions start a new scope

```
> function increment(x) {
    return x + 1;
}
> increment(5)
6
```

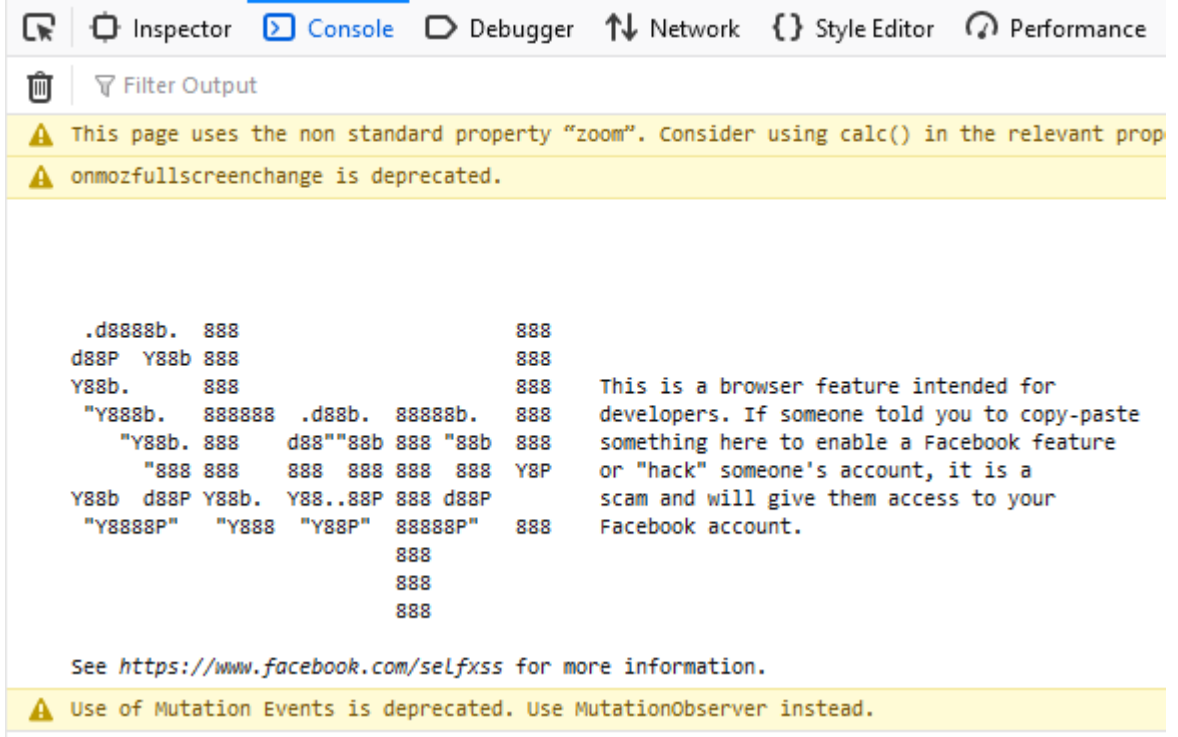
JavaScript concept: functions

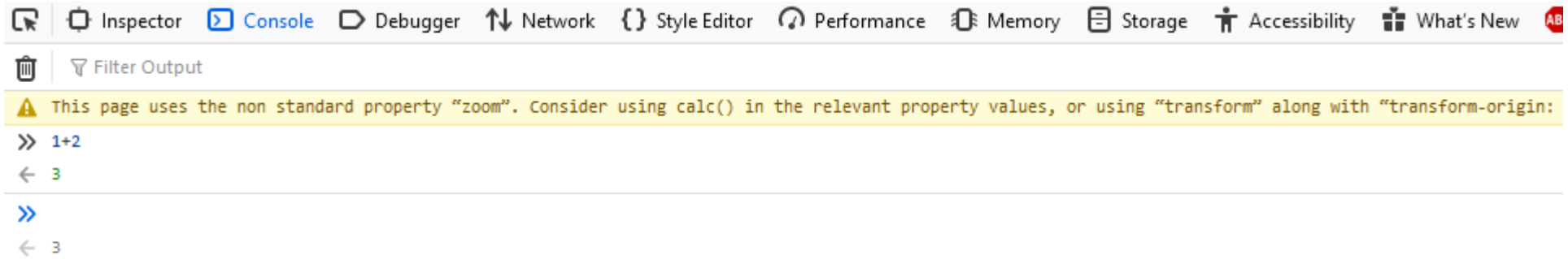
- JavaScript functions are callable objects bound to a name
- JavaScript functions are garbage collected
- JavaScript functions execute in an *activation record*
- Activation records are also garbage collected

```
> function increment(x) {
    return x + 1;
}
> increment(5)
6

> let fn = increment
> fn(5)
6
```

Console

- Way to log things you don't need users to see
- Visible in browser developer tools





Outline

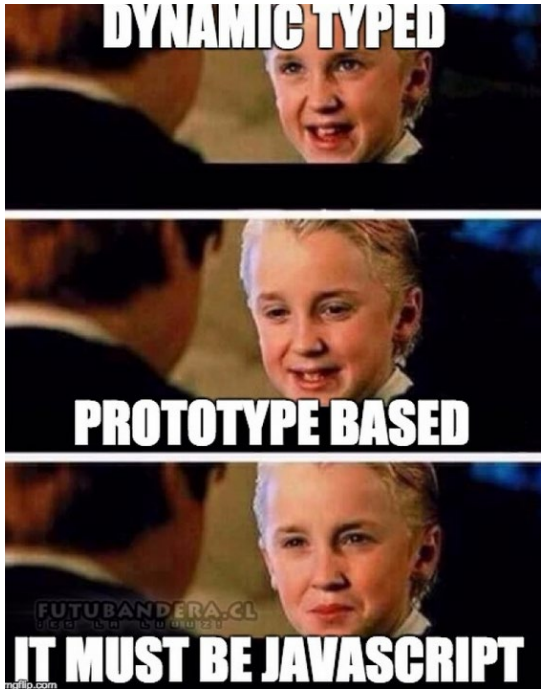
- Review: safety and idempotency
- Intro to JavaScript
- **Prototypes**
- Event-Driven Programming

Prototypes

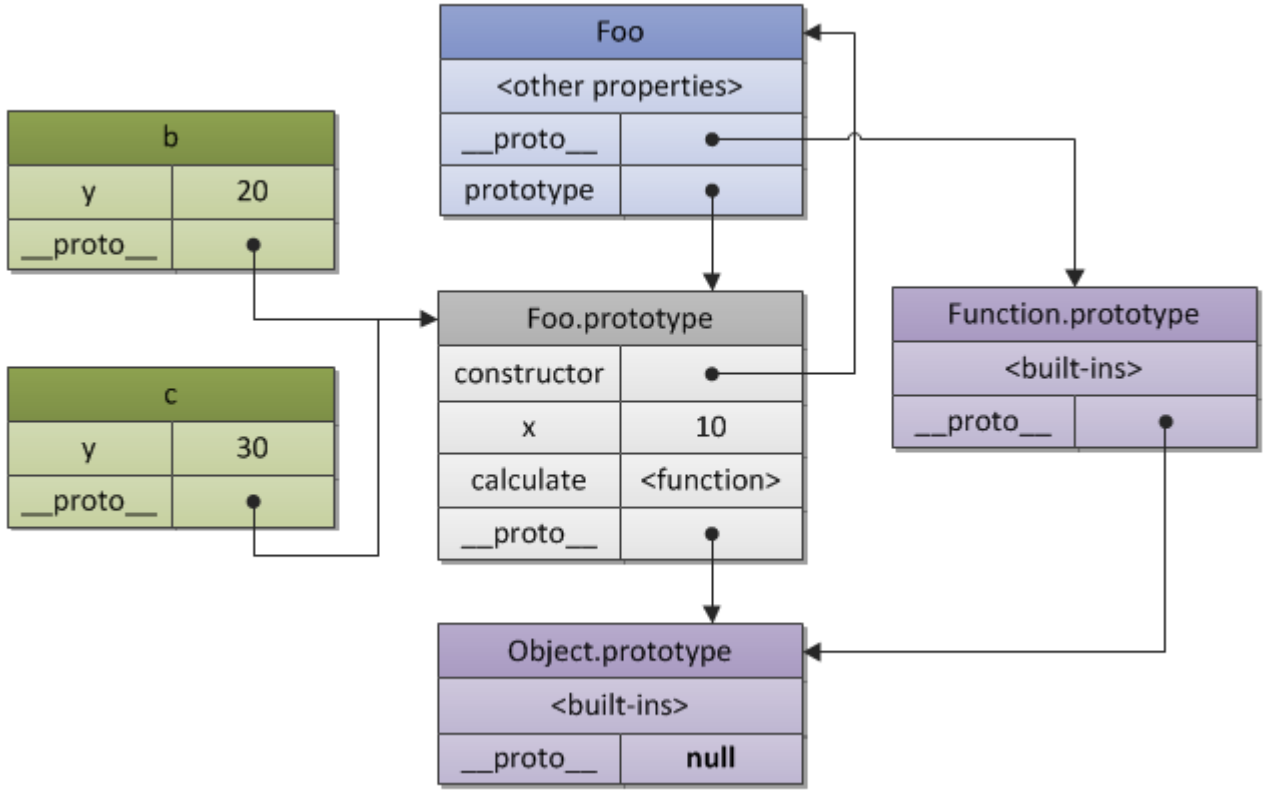
- JavaScript is a “prototypical object-oriented language”
- *Objects* have *properties* and a *prototype*
- *Properties* are values associated with an object
- *Prototypes* are the mechanism by which JavaScript objects inherit features from one another
- In JavaScript, there is no distinction between instances and classes/types
  - Everything is an object
- For Java/C/Python programmers, prototypes feel very strange

Prototypes

- Every JS object has a *prototype attribute*
  - Akin to the object's "parent"
  - All objects inherit the properties and methods from their prototype
    - When resolving a reference, JS climbs prototype tree until name is found (or not)
  - The prototype is *another object*, not a superclass
  - Examine it via the `__proto__` attribute
  - **CAREFUL (also: WEIRD)**: `__proto__` and `prototype` are not the same thing!



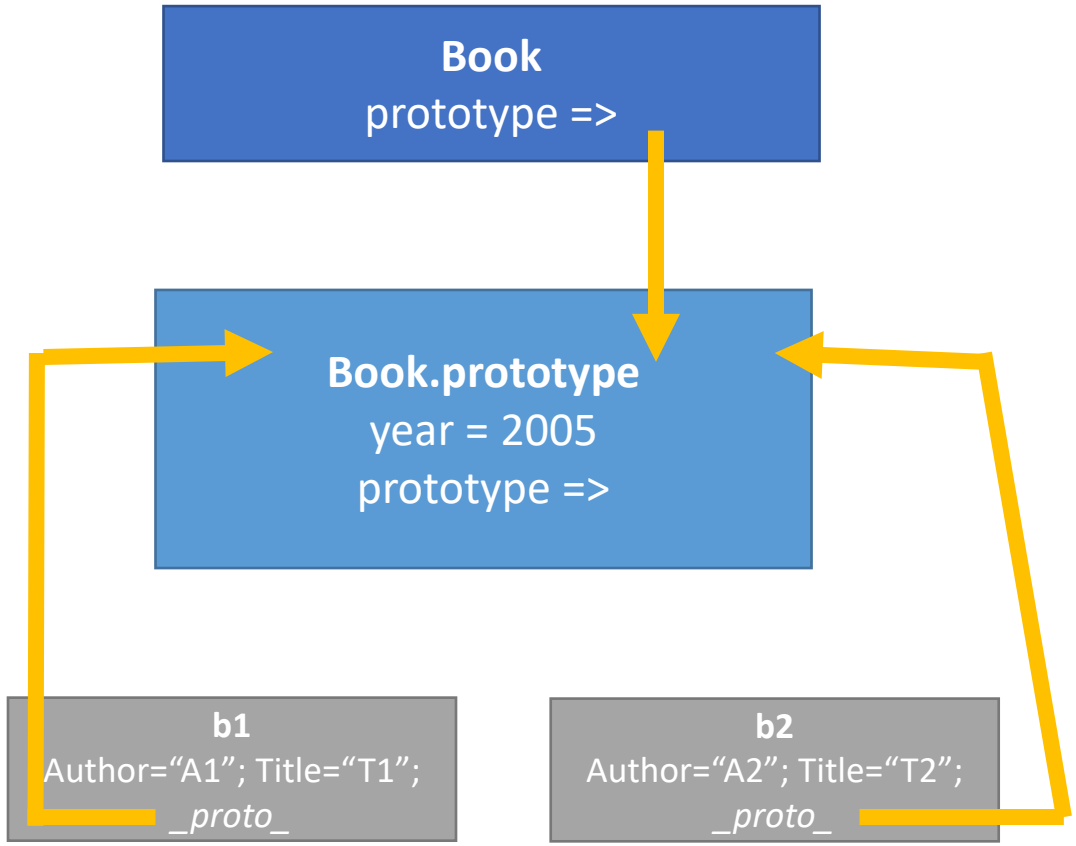
Prototypes




```
Function Foo(val){
 this.y = val;
}
let b = new Foo(20);
let c = new Foo(30);

Foo.prototype.x = 10;
Foo.prototype.calculate = function (z) {
 return this.x + this.y + z;
}
console.log( b.calculate(30) );
console.log( c.calculate(40) );
```

Prototype picture

```
function Book(author, title) {
   this.author = author;
   this.title = title;
   // ...
}

let b1 = new Book("A1", "T1");
let b2 = new Book("A2", "T2");

b1.pages = 100;
Book.prototype.year = 2005;
```



Outline

- Review: safety and idempotency
- Intro to JavaScript
- Prototypes
- **Event-Driven Programming**

Event-driven programming

- In event-driven programming, the flow of the program is determined by *events*

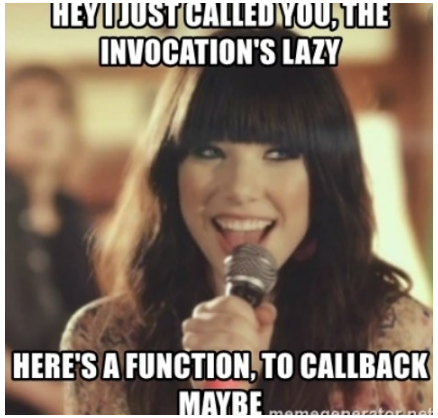
- A few examples of events built into the browser:
  - `onclick`: user clicks a button
  - `onmouseover`: The user moves the mouse over an HTML element
  - `onkeydown`: The user pushes a keyboard key
  - `onload`: The browser has finished loading the page

- Event-driven programming is useful for GUIs like web applications
  - Windows works like this too

Callback functions

- A main loop listens for events and triggers a *callback function*
- A callback function is just a normal function, waiting to be executed
  - Current example: `hello()` is a function that can be *called back* later

```
function hello() {
  document.getElementById("JSEntry").innerHTML =
    "Hello World!";
}
```



Event handlers

- In the HTML, we registered our function as an *event handler*
- That means telling the browser "please run this function when X event occurs"

```
<button onClick="hello()" type="button">
  Click Me!
</button>
```

- The JavaScript interpreter maintains a table of events that map to functions

Event	Function
onClick	hello

The event queue

- In JavaScript, function calls live on the stack, objects live on the heap, and *messages live on the queue*
- The function on the top of the stack executes.
- *When the stack is empty, a message is taken out of the queue and processed.*
- Each message is a function

- An event adds a message to the queue

Event-driven programming: overview 2 example

```
function world() {
    let b1 = new Book("A1", "T1");
}
function hello() {
    world();
}
```

Placing events on queue

- Browser events like clicks go on queue
- setTimeout(): a function to add an event to the queue

```
function runLater() {
    console.log("Ran!");
}
setTimeout(runLater, 1000);
```

Exam-style Exercise

- What is the output of this code?

```
function f() {
  console.log(‘beginning');
  function callback1() {
    console.log('callback1');
  }
  setTimeout(callback1, 1000); //1s
  console.log('middle');
  function callback2() {
    console.log('callback2');
  }
  setTimeout(callback2, 2000); //2s
  console.log('end');
}
f();
```

Outline

- Review: safety and idempotency
- Intro to JavaScript
- Prototypes
- Event-Driven Programming
- **Backup:** common pitfalls

Common mistake: equality operators

- “JavaScript has two sets of equality operators: === and !==, and their evil twins == and !=.” -- Douglas Crockford
- == performs a type conversion when comparing two things
- === no type conversion
  - Return false if the types differ

ALWAYS use === and !==

- A few interesting cases:

```
'' == '0' // false
0 == '' // true
0 == '0' // true
```

Common mistake: scope

- "Simply" assigning values always creates a global variable

```
> function f() {
    x = 5;
  }
> f();
> x
5
```

NEVER simply assign values

Common mistake: scope

- `var` creates a local or global scoped variable
- Functions create scope

```
> var x = 0;
> function f() {
    var x = 5;
  }
> f();
> x
0
```

- Other blocks do not

```
> var x = 0;
> if (x === 0) {
    var x = 5;
  }
> x
5
```

Common mistake: hoisting

- Variables declared with `var` are *hoisted* to the top of the function

```
> function f() {
    console.log(x === undefined);
    var x = 5;
  }
> f();
true
```

- Variables declared with `let` or `const` are not

```
> function f() {
    console.log(x === undefined);
    let x = 5;
  }
> f();
ReferenceError: x is not defined
```

Common misunderstanding: `const`

- `const` means you can't reassign the reference

```
> const eecs485 = { name: 'Web Systems', num: 485 };
> eecs485 = { name: 'Chicken Stories', num: 101 };
TypeError: Assignment to constant variable.
```

- Changing the object is OK

```
> const eecs485 = { name: 'Web Systems', num: 485 };
> eecs485.name = 'Chicken Stories';
'Chicken Stories'
> eecs485.num = 101;
101
> eecs485
{ name: 'Chicken Stories', num: 101 }
```

- `const x` in JavaScript is like `int *const p` in C.

Common mistake: for-in loops

- `for-in` loops often yield unexpected results
  - They iterate "up the prototype chain"

```
> const chickens = ['Magda', 'Marilyn', 'Myrtle II'];
> for (let chicken in chickens) {
>   console.log(chicken);
> }
1
2
3
```

- ES6's for-of loops are nice, but are hard to analyze statically, so some style guides do not allow them

```
> for (let chicken of chickens) {
>   console.log(chicken);
> }
Magda
Marilyn
Myrtle II
```

Iteration with `forEach` and `map`

- `forEach` loops "do the right thing"
  - Behave like other programming languages (C, C++, Perl, Python ...)
  - We'll learn about the => syntax soon (it's an anonymous function)

```
const chickens = ['Magda', 'Marilyn', 'Myrtle II'];

chickens.forEach((chicken) => {
  console.log(chicken);
});
```

- `map` is another nice option
  - Use it to transform an array into another array

```
const chickens_say = chickens.map(chicken => (
  `${chicken} says cluck`
));
console.log(chickens_say);
//[ 'Magda says cluck', 'Marilyn says cluck',
//  'Myrtle II says cluck' ]
```