

Classes
vs
Simple Object Delegation
In JavaScript

Things to consider...

Do we want to (deeply) understand what our code is doing?

Do we want encapsulation?

Do we want true private variables and functions?

**Let's get started with some
concepts...**


Typically, JavaScript is described in terms of having a **prototypal inheritance** model.

However, this has caused a lot of **confusion**. Why?

- JavaScript *is* prototypal in nature, meaning its behavior is based upon objects acting as prototypes of other objects...
- *But*, it doesn't inherit or copy down behavior and attributes from an ancestor. That is, there is no parent/child relationship.
- *Rather*, there is a live link between the objects.
- Objects *may* be related to each other, but there is no rigid taxonomy like there is in class-based languages that intrinsically defines these relationships.


Inheritance

```
class Bar {}  
  
var foo = new Bar()
```



Delegation

```
{ bar }  
  
var foo = Object.create(bar)
```



So, prototypal inheritance can be considered
an **oxymoron**.

Let's think of it as **delegation**, instead.

How It Works

If an object doesn't have a function or variable, it will delegate, or ask, the previous object on the prototype chain if it can satisfy the query.

This process will repeat until the end of the prototype chain is reached.

Ok, so what is the **prototype chain**?

The prototype chain is the aggregate of the hidden links between objects.



The arrows between the objects show the direction of the delegation.

These are hidden links that point to the previous object in the chain, which serves as its `[[Prototype]]`.

The links are internal properties created by the JavaScript engine, although some implementations do expose this link as the `__proto__` attribute on objects.

Note that this hidden link may point to the same object as `[[Constructor]].prototype`, but not always! It depends on how the object was created.

Creating Objects

{}

- Object literal syntax
- The new object delegates to `Object.prototype`

The *new* operator

- **Indirectly** creates a new object and returns it as the result of a constructor call
- The new object delegates to `[[Constructor]].prototype`
- “Classical Inheritance” - a pattern to imitate class-based languages

`Object.create()`

- **Directly** creates a new object and returns it
- The new object delegates to the passed object which becomes the new object's `[[Prototype]]`
- Delegation - Objects linked to other objects

Speaking of “**classical inheritance**”, isn’t that
the same as **object delegation**?

No. There is **no such thing as** “classical inheritance” in JavaScript.

When developers refer to “classical inheritance”, they are really describing a pattern that could be called the **constructor call** pattern.

However, delegation is not a pattern, it is actually **how the language works!**

Why is this important to understand?

Delegation is the exact opposite of inheritance,
and not understanding the differences and ramifications *will* lead to
hard to understand **bugs**.

And features of the language will seem mysterious and magical...

instanceof

with

Can I haz code?

Simple Object Delegation

```
const base = {
  getCoords() {
    return {
      x: this.x,
      y: this.y
    };
  },

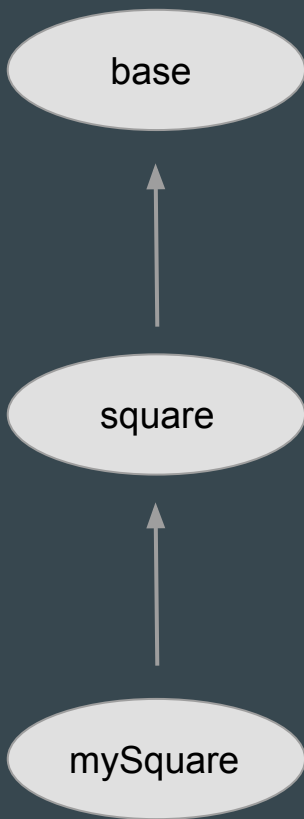
  getType() {
    return this.type || 'base';
  }
};

const square = Object.create(base, {
  type: { value: 'square' }
});
```

```
// Let's setup our delegation.
const mySquare = Object.create(square, {
  x: { value: 100 },
  y: { value: 100 }
});

mySquare.getType(); // 'square'
mySquare.getCoords(); // { x: 100, y: 100 }
```

Objects linked to other objects! Neat!



Programmers love abstractions! Here's one.

```
const delegate = (proto, ...rest) => {  
  const obj = Object.create(proto);  
  
  return rest.reduce((prev, curr) =>  
    Object.assign(obj, curr)  
  , obj);  
};
```

Simple Object Delegation Revisited

```
const base = {  
  getCoords() {  
    return {  
      x: this.x,  
      y: this.y  
    };  
  },  
  
  getType() {  
    return this.type || 'base';  
  }  
};
```

```
const square = delegate(base, {  
  type: 'square'  
});
```

```
// Let's setup our delegation.  
const mySquare = delegate(square, {  
  x: 100,  
  y: 100  
});  
  
mySquare.getType(); // 'square'  
mySquare.getCoords(); // { x: 100, y: 100 }
```


That's pretty simple!



INDEED

**But alas, it was not meant
to be...**

In the mid-aughts, JavaScript libraries and frameworks began emerging, and the thought was to make them “look more like Java” to gain adoption.

We need classes!

Quick aside...

Two Thoughts...

- Isn't that a really long time ago?
- Isn't it time we revisit (and rethink) that?

Ok, moving on...

Classical inheritance

```
function Base(type) {
  this.type = type || 'base';
}

Base.prototype.getCoords = function () {
  return {
    x: this.x,
    y: this.y
  };
};

Base.prototype.getType = function () {
  return this.type;
};
```

```
function Square(x, y) {
  Base.call(this, 'square');

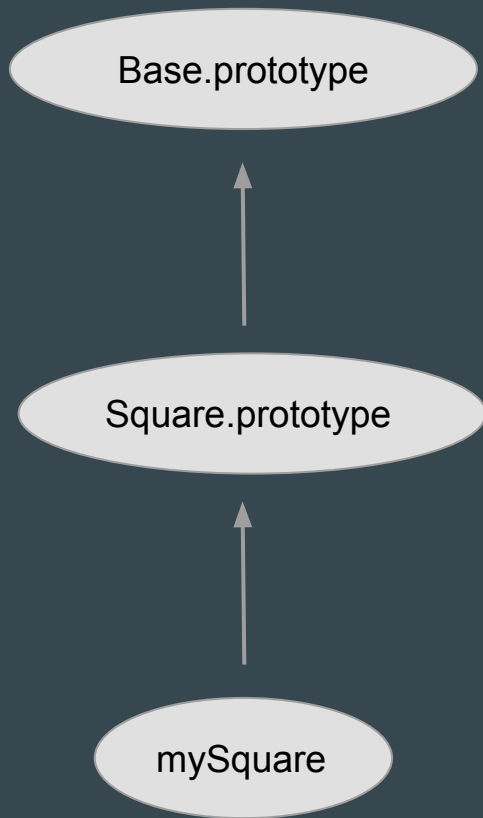
  this.x = x;
  this.y = y;
}

Square.prototype = new Base();
Square.prototype.constructor = Square;
```

```
// Let's setup our delegation.
const mySquare = new Square(100, 100);

mySquare.getType(); // 'square'
mySquare.getCoords(); // { x: 100, y: 100 }
```


Objects linked to other objects! Neat!



Yay, I see a `new` operator, and things are capitalized! And is that supposed to be a constructor function?!? But...

That's kind of gross, and I don't need or want to know all that 'prototype' stuff. Can we clean that up?

ES2015

ES2015 class syntax

```
class Base {
  constructor(type) {
    this.type = type || 'base';
  }

  getCoords() {
    return {
      x: this.x,
      y: this.y
    };
  }

  getType() {
    return this.type;
  }
}
```

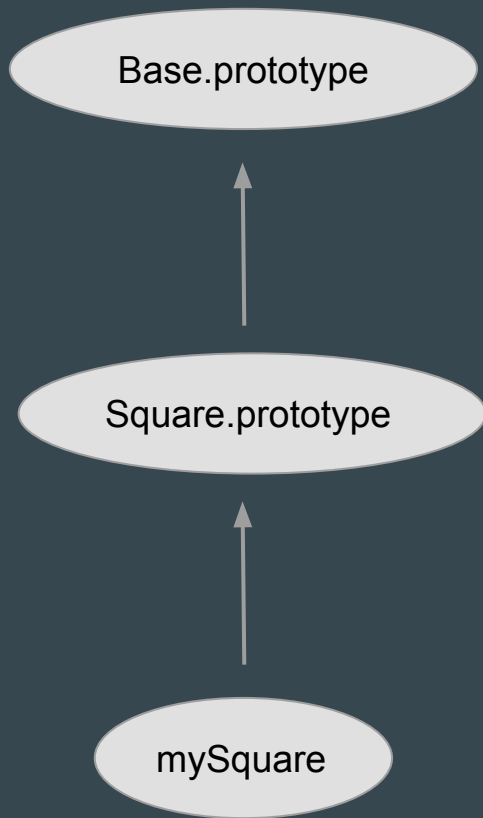
```
class Square extends Base {
  constructor(x, y) {
    super('square');

    this.x = x;
    this.y = y;
  }
}
```

```
// Let's setup our delegation.
const mySquare = new Square(100, 100);

mySquare.getType(); // 'square'
mySquare.getCoords(); // { x: 100, y: 100 }
```

Objects linked to other objects! Neat!



If prototype properties are desired, encapsulation is broken.

...

```
class Square extends Base {  
  constructor(x, y) {  
    super('square');  
  
    this.x = x;  
    this.y = y;  
  }  
}
```

```
Square.prototype.total = 0;
```

Hey, wait, where'd that 'prototype' come from? You promised I could avoid that in ES2015 with classes!

Furthermore, how would I have known I could use that? There's nothing else in the code that indicates it!

So...it must be magic!

And there be dragons.

...

```
class Square extends Base {
  constructor(x, y) {
    super('square');

    this.x = x;
    this.y = y;

    this.total++;
  }
}
```

```
Square.prototype.total = 0;
```

```
// Let's setup our delegation.
const mySquare = new Square(100, 100);

mySquare.total; // 1
mySquare.hasOwnProperty('total'); // true!
Square.prototype.total; // 0...what?!?
```


Of course there's a fix, but it stinks.

...

```
class Square extends Base {
  constructor(x, y) {
    super('square');

    this.x = x;
    this.y = y;

    Square.prototype.total++;
  }
}
```

```
Square.prototype.total = 0;
```

```
// Let's setup our delegation.
const mySquare = new Square(100, 100);

mySquare.total; // 1
mySquare.hasOwnProperty('total'); // false
Square.prototype.total; // 1

// Great, now I'm back to having to use
// and know 'prototype' again! :(
```

The End