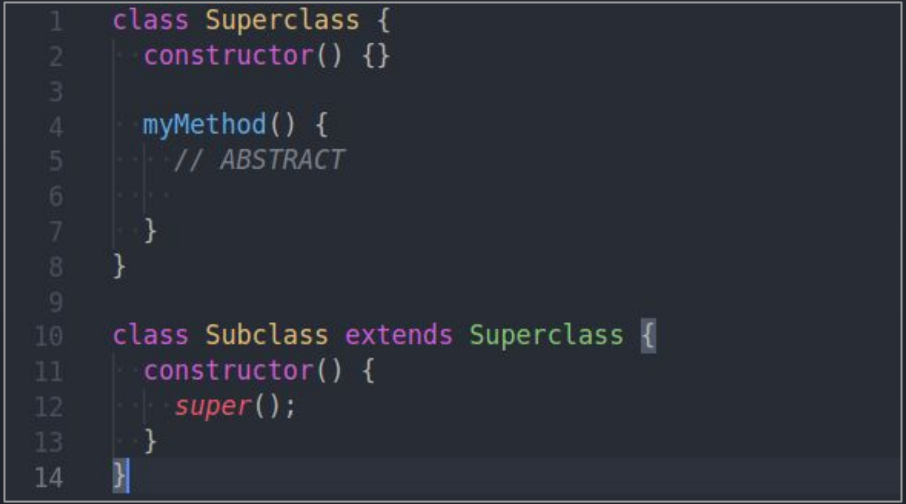
JavaScript Design Patterns

Abstract Methods, Accessors, and the Module Pattern

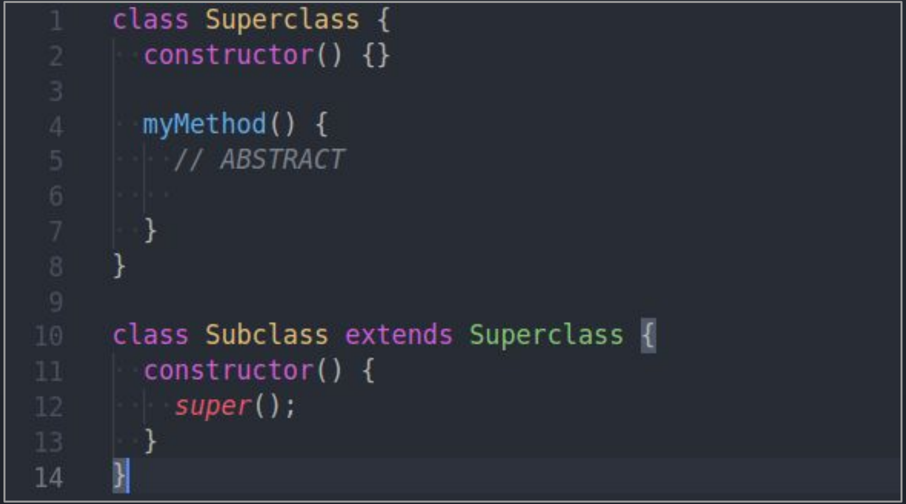
Andrew Burks
https://github.com/AndrewTBurks/CS474_HW04

Enforcing Abstract Methods

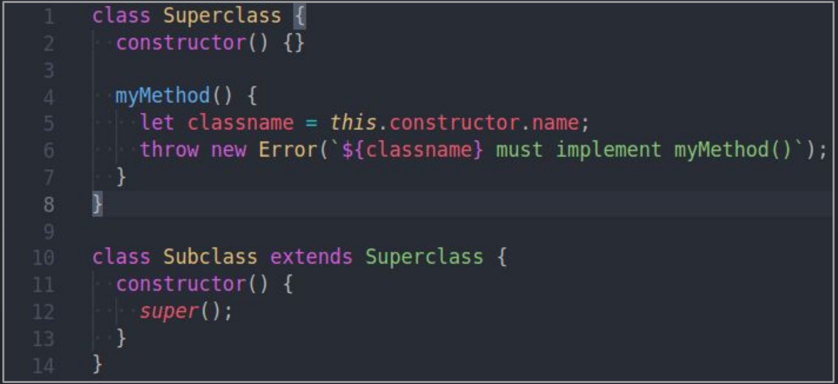


```
class Superclass {
  constructor() {}

  myMethod() {
    // ABSTRACT

  }
}

class Subclass extends Superclass {
  constructor() {
    super();
  }
}
```

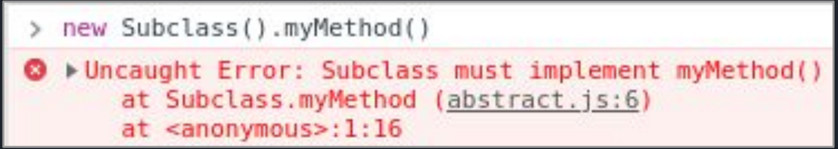
Enforcing Abstract Methods



```javascript
class Superclass {
  constructor() {}

  myMethod() {
    // ABSTRACT

  }
}

class Subclass extends Superclass {
  constructor() {
    super();
  }
}
```



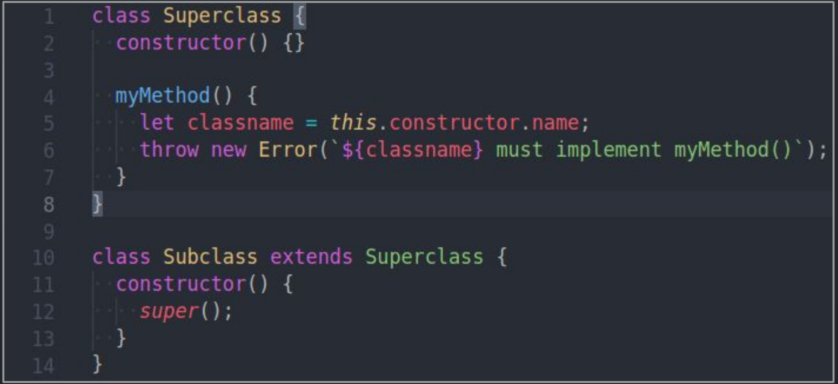
```javascript
class Superclass {
  constructor() {}

  myMethod() {
    let classname = this.constructor.name;
    throw new Error(`${classname} must implement myMethod()`);
  }
}

class Subclass extends Superclass {
  constructor() {
    super();
  }
}
```



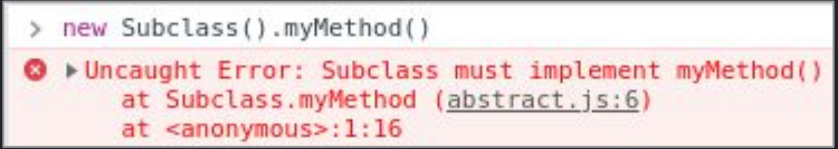
```
> new Subclass().myMethod()
✖ ▶Uncaught Error: Subclass must implement myMethod()
      at Subclass.myMethod (abstract.js:6)
      at <anonymous>:1:16
```

Enforcing Abstract Methods

- `throw new Error()` can be used to enforce that abstract methods be implemented
- This allows for a runtime check of implementation
- The `this.constructor.name` syntax can be used to identify the offending subclass

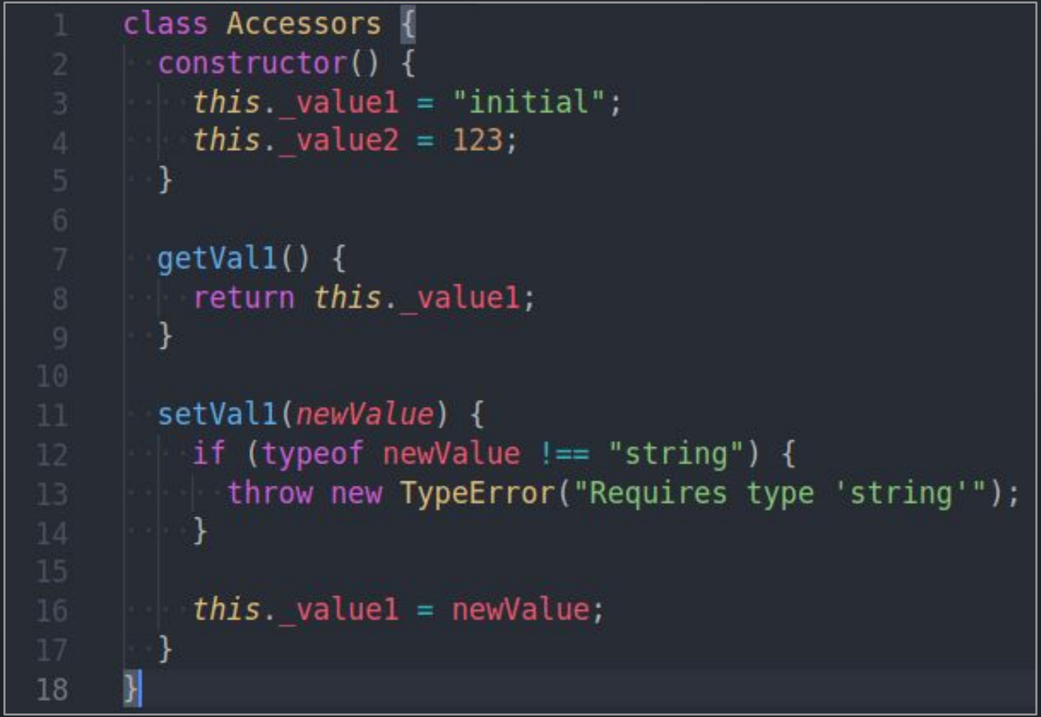


```javascript
class Superclass {
  constructor() {}

  myMethod() {
    let classname = this.constructor.name;
    throw new Error(`${classname} must implement myMethod()`);
  }
}

class Subclass extends Superclass {
  constructor() {
    super();
  }
}
```



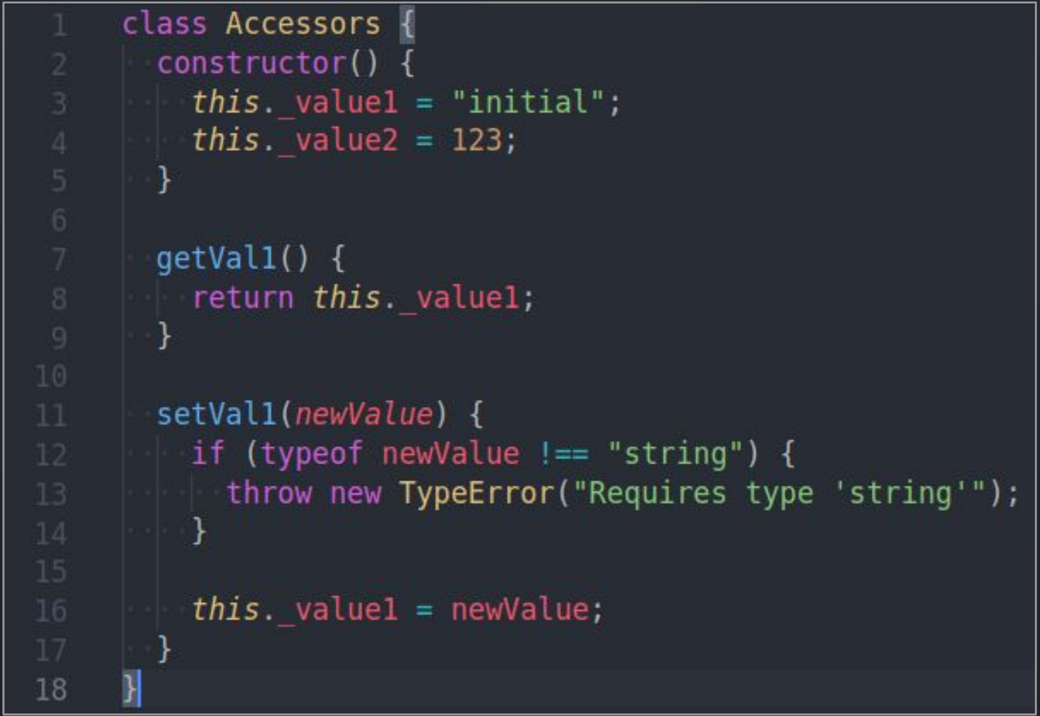
```
> new Subclass().myMethod()
✖ ▶Uncaught Error: Subclass must implement myMethod()
      at Subclass.myMethod (abstract.js:6)
      at <anonymous>:1:16
```

Property Accessors

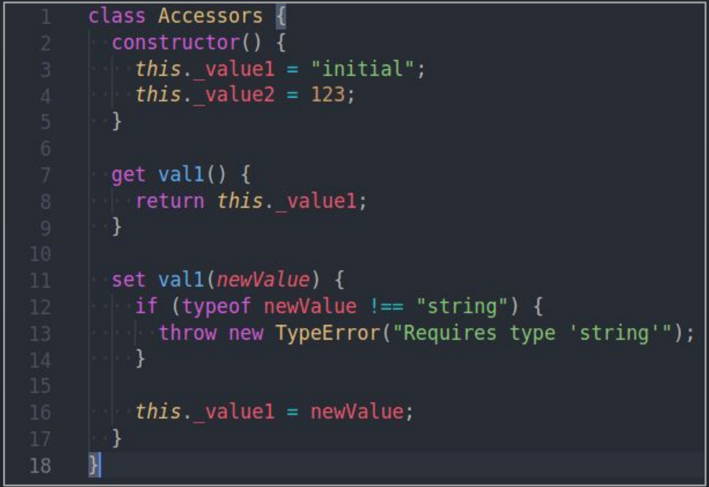


```
class Accessors {
  constructor() {
    this._value1 = "initial";
    this._value2 = 123;
  }

  getVal1() {
    return this._value1;
  }

  setVal1(newValue) {
    if (typeof newValue !== "string") {
      throw new TypeError("Requires type 'string'");
    }

    this._value1 = newValue;
  }
}
```

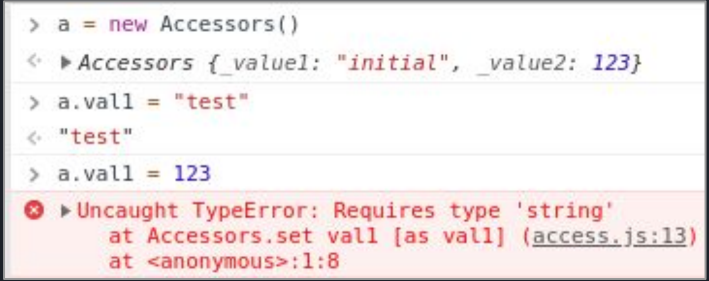
Property Accessors



```javascript
class Accessors {
  constructor() {
    this._value1 = "initial";
    this._value2 = 123;
  }

  getVal1() {
    return this._value1;
  }

  setVal1(newValue) {
    if (typeof newValue !== "string") {
      throw new TypeError("Requires type 'string'");
    }

    this._value1 = newValue;
  }
}
```



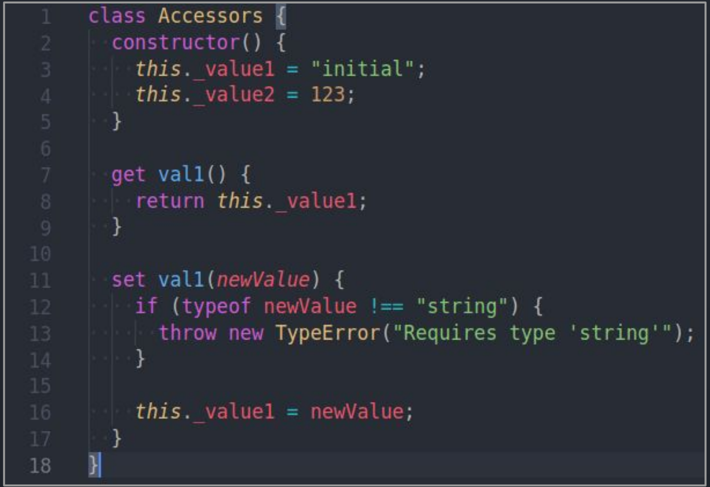
```javascript
class Accessors {
  constructor() {
    this._value1 = "initial";
    this._value2 = 123;
  }

  get val1() {
    return this._value1;
  }

  set val1(newValue) {
    if (typeof newValue !== "string") {
      throw new TypeError("Requires type 'string'");
    }

    this._value1 = newValue;
  }
}
```



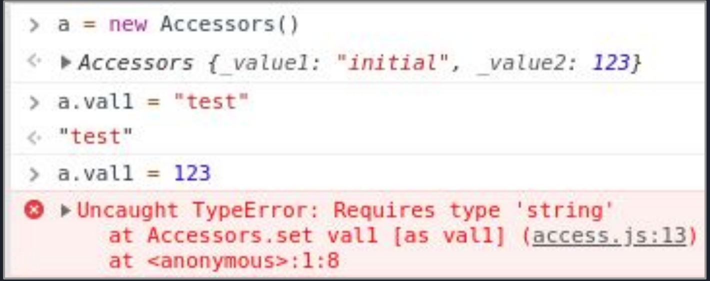
```
> a = new Accessors()
<· ▶Accessors {_value1: "initial", _value2: 123}
> a.val1 = "test"
<· "test"
> a.val1 = 123
✖ ▶Uncaught TypeError: Requires type 'string'
     at Accessors.set val1 [as val1] (access.js:13)
     at <anonymous>:1:8
```

Property Accessors

- `get/set` property accessors simplify interaction with object properties
- Properties may be accessed normally using dot-notation (`obj.val1 = …`)
- However, more protection seen from traditional getters/setters can be implemented around the access
- This can help with **observable/reactive** patterns



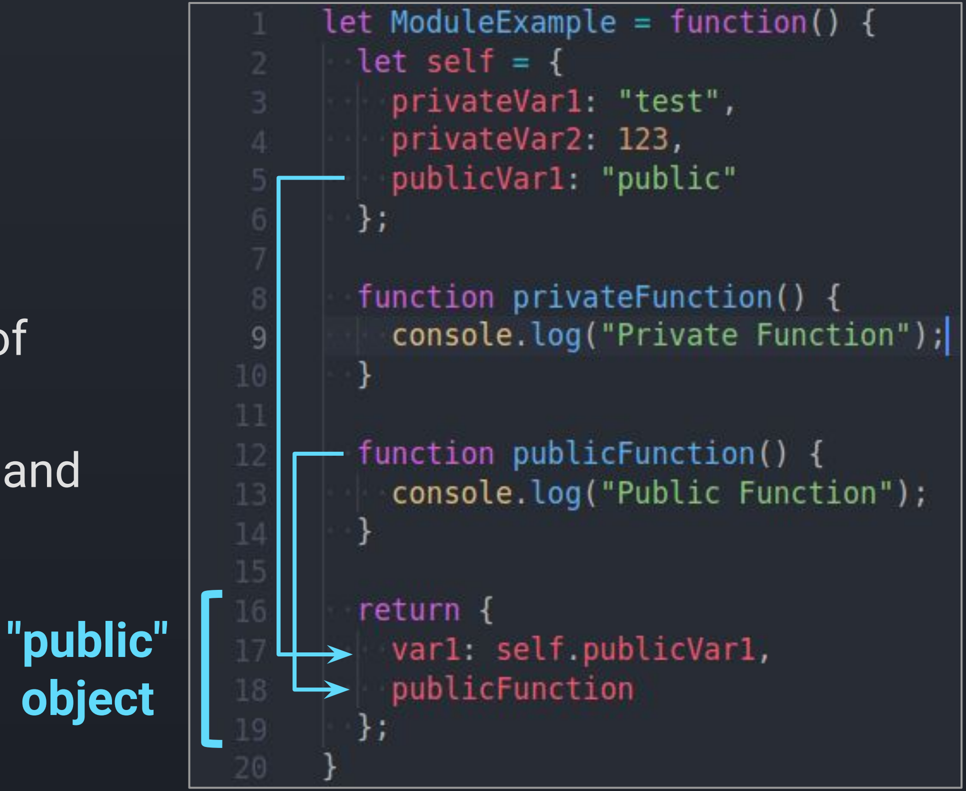
```javascript
class Accessors {
  constructor() {
    this._value1 = "initial";
    this._value2 = 123;
  }

  get val1() {
    return this._value1;
  }

  set val1(newValue) {
    if (typeof newValue !== "string") {
      throw new TypeError("Requires type 'string'");
    }

    this._value1 = newValue;
  }
}
```



```
> a = new Accessors()
<· ▶Accessors {_value1: "initial", _value2: 123}
> a.val1 = "test"
<· "test"
> a.val1 = 123
✖ ▶Uncaught TypeError: Requires type 'string'
      at Accessors.set val1 [as val1] (access.js:13)
      at <anonymous>:1:8
```

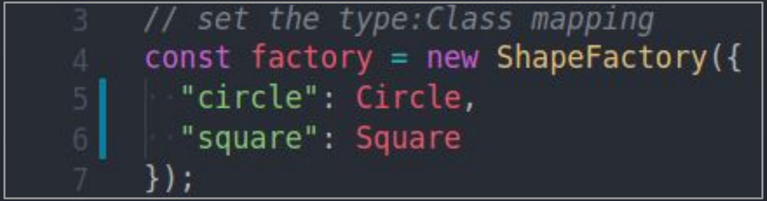
The Revealing Module Pattern

- **Scoping/closure** allows for specification of private or public variables
- Don't need to worry about scope of *"this"*
- Function names within the object and in the interface can be different

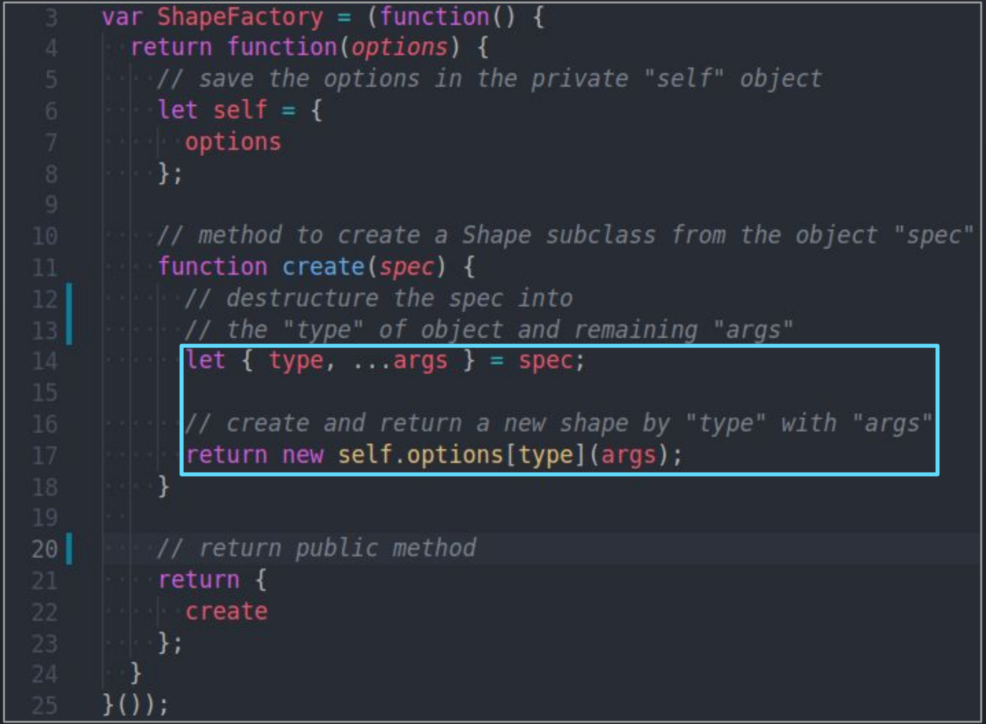


```
let ModuleExample = function() {
  let self = {
    privateVar1: "test",
    privateVar2: 123,
    publicVar1: "public"
  };

  function privateFunction() {
    console.log("Private Function");
  }

  function publicFunction() {
    console.log("Public Function");
  }

  return {
    var1: self.publicVar1,
    publicFunction
  };
}
```

Bonus: Module Factory for Shapes



```
// set the type:Class mapping
const factory = new ShapeFactory({
  "circle": Circle,
  "square": Square
});
```



```
var ShapeFactory = (function() {
  return function(options) {
    // save the options in the private "self" object
    let self = {
      options
    };

    // method to create a Shape subclass from the object "spec"
    function create(spec) {
      // destructure the spec into
      // the "type" of object and remaining "args"
      let { type, ...args } = spec;

      // create and return a new shape by "type" with "args"
      return new self.options[type](args);
    }

    // return public method
    return {
      create
    };
  }
}());
```