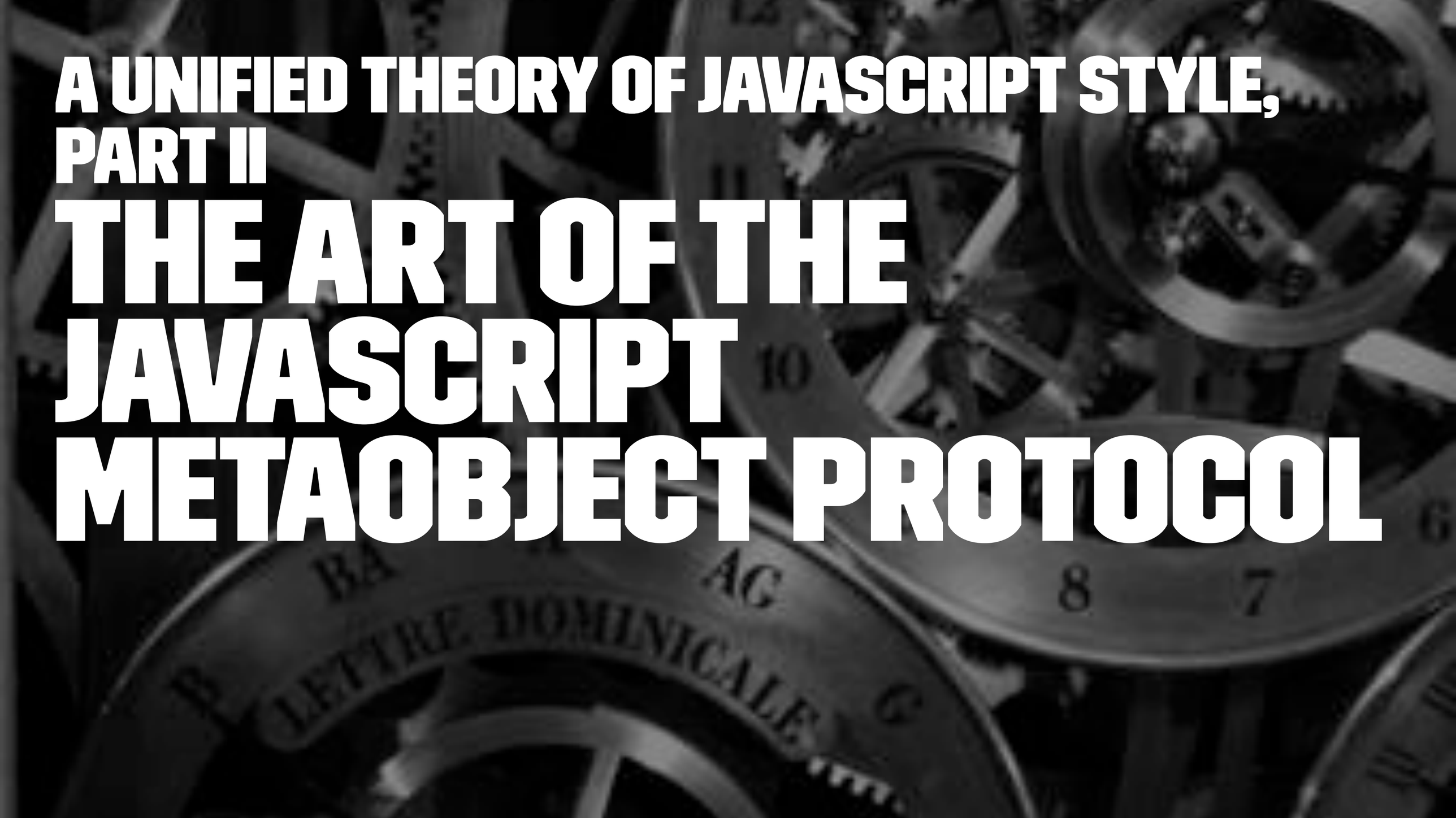

12
11
10
9
8
7
6
BA
A
AG
B
LETTRE DOMINICALE
G



A UNIFIED THEORY OF JAVASCRIPT STYLE, PART II

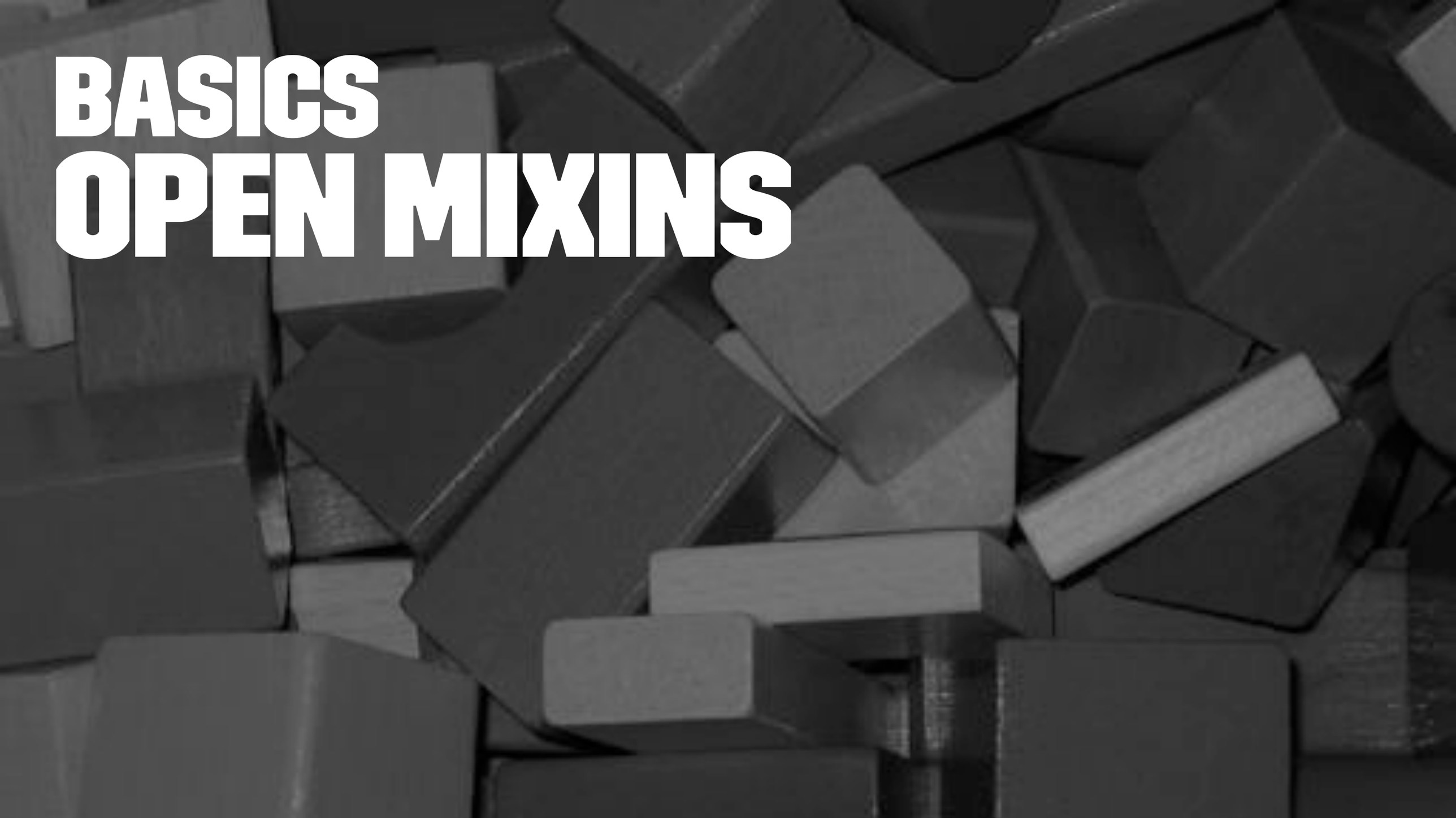
THE ART OF THE JAVASCRIPT METAOBJECT PROTOCOL


ON THE AIR


WE'LL TALK ABOUT
PROXIES,
ENCAPSULATION, AND
COMPOSITION


BUT THINK ABOUT
FLEXIBILITY AND
DECOUPLING AT SCALE

BASICS
OPEN MIXINS



```
var sam = {
  firstName: 'Sam',
  lastName: 'Lowry',
  fullName: function () {
    return this.firstName + " " + this.lastName;
  },
  rename: function (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
}
```

```
var sam = {
  firstName: 'Sam',
  lastName: 'Lowry'
};

var Person = {
  fullName: function () {
    return this.firstName + " " + this.lastName;
  },
  rename: function (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
};
```

```
function extend () {
  var consumer = arguments[0],
      providers = [].slice.call(arguments, 1),
      key,
      i,
      provider;

  for (i = 0; i < providers.length; ++i) {
    provider = providers[i];
    for (key in provider) {
      if (provider.hasOwnProperty(key)) {
        consumer[key] = provider[key];
      };
    };
  };
  return consumer;
}
```

MIXINS ARE MANY TO _

```
extend(sam, Person);

var peck = {
  firstName: 'Sam',
  lastName: 'Peckinpah'
};

extend(peck, Person);
```

MIXINS ARE _ TO MANY

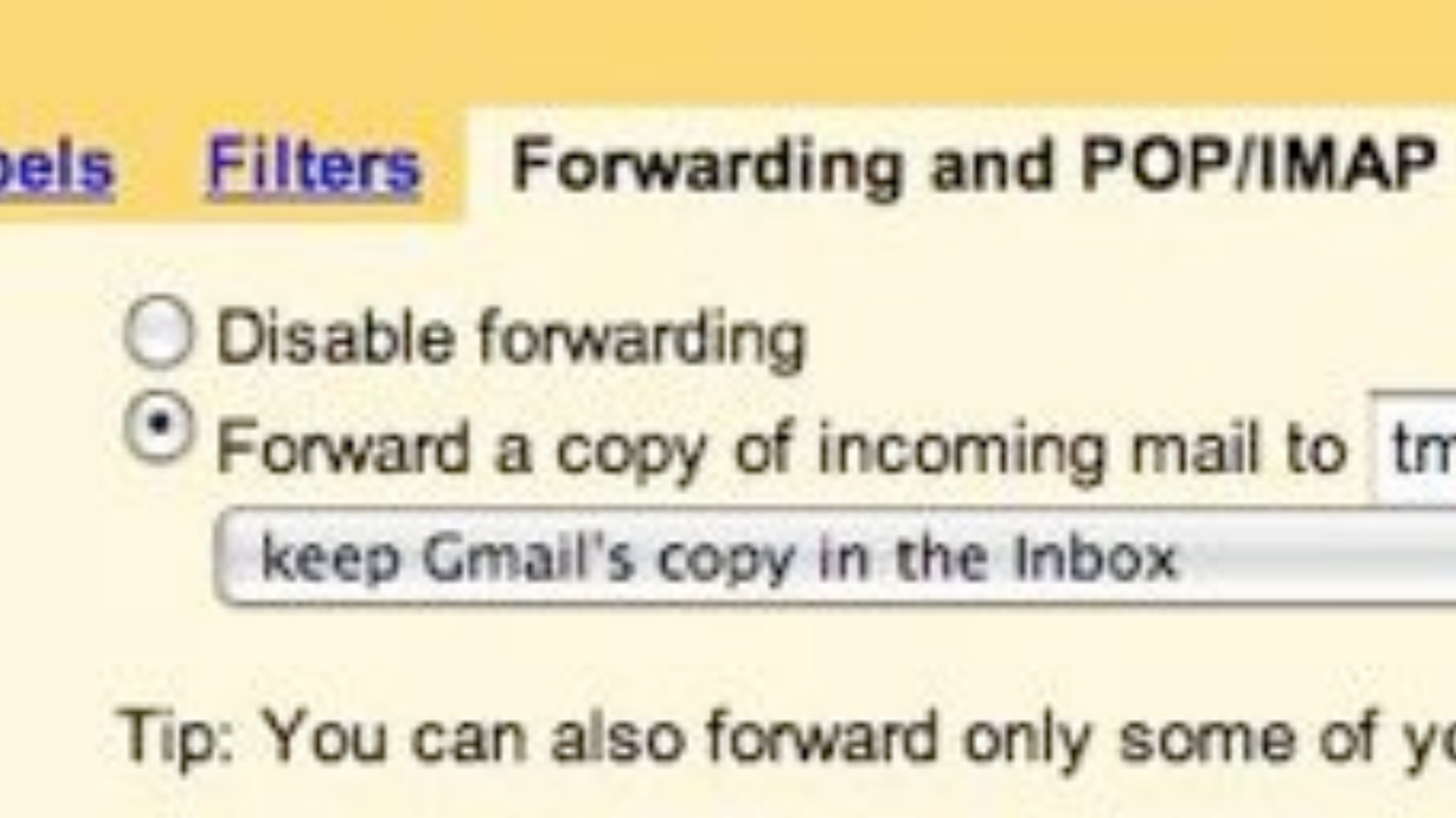
```
var HasCareer = {
  career: function () {
    return this.chosenCareer;
  },
  setCareer: function (career) {
    this.chosenCareer = career;
    return this;
  }
};

extend(peck, HasCareer);

peck.setCareer('Director');
```


NO
ENTRY

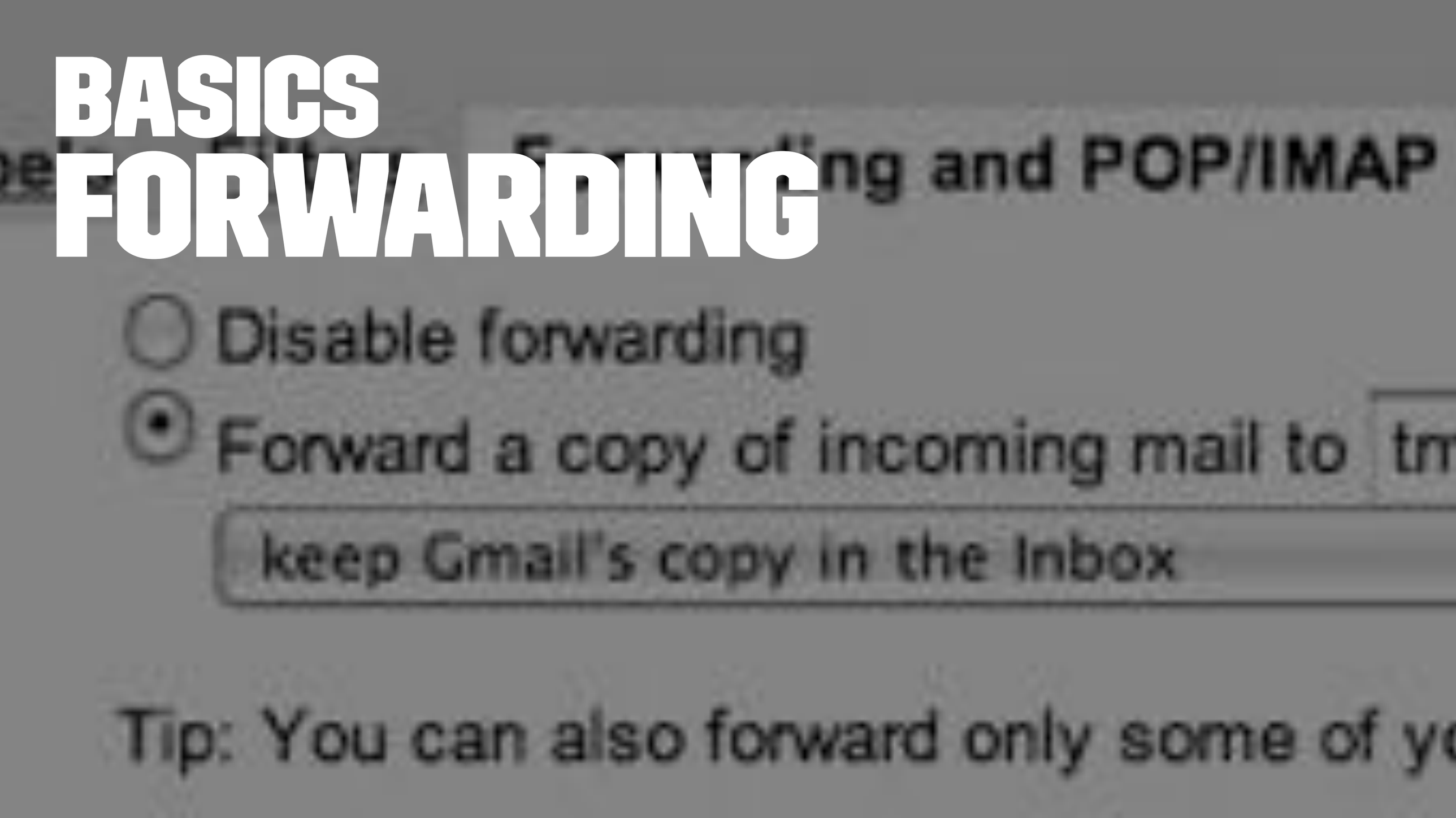
BASICS
PRIVATE MIXINS

```
function extendPrivately (receiver, mixin) {
  var methodName,
      privateProperty = Object.create(null);

  for (methodName in mixin) {
    if (mixin.hasOwnProperty(methodName)) {
      receiver[methodName] = mixin[methodName].bind(privateProperty);
    };
  };
  return receiver;
};
```

```
var peck = {
  firstName: 'Sam',
  lastName: 'Peckinpah'
};

extendPrivately(peck, HasCareer);

peck.setCareer("Director");

peck.chosenCareer
    //=> undefined
```


bels
Filters
Forwarding and POP/IMAP
Disable forwarding
Forward a copy of incoming mail to
keep Gmail's copy in the Inbox
Tip: You can also forward only some of

BASICS
FORWARDING



```
function forward (receiver, metaobject, methods) {
  if (methods == null) {
    methods = Object.keys(metaobject).filter(function (methodName) {
      return typeof(metaobject[methodName]) == 'function';
    });
  }
  methods.forEach(function (methodName) {
    receiver[methodName] = function () {
      var result = metaobject[methodName].apply(metaobject, arguments);
      return result === metaobject ? this : result;
    };
  });

  return receiver;
};
```


FORWARDING IS IMPORTANT
LET'S STICK A TACK IN
THIS FOR LATER

FOUR SHADES OF GRAY

1. A mixin uses the receiver's method body, and executes in the receiver's context.
2. A private mixin uses the receiver's method body, but executes in another object's context.
3. Forwarding uses another object's method body, and executes in another object's context.
4. What uses another object's method body, but executes in the receiver's context?

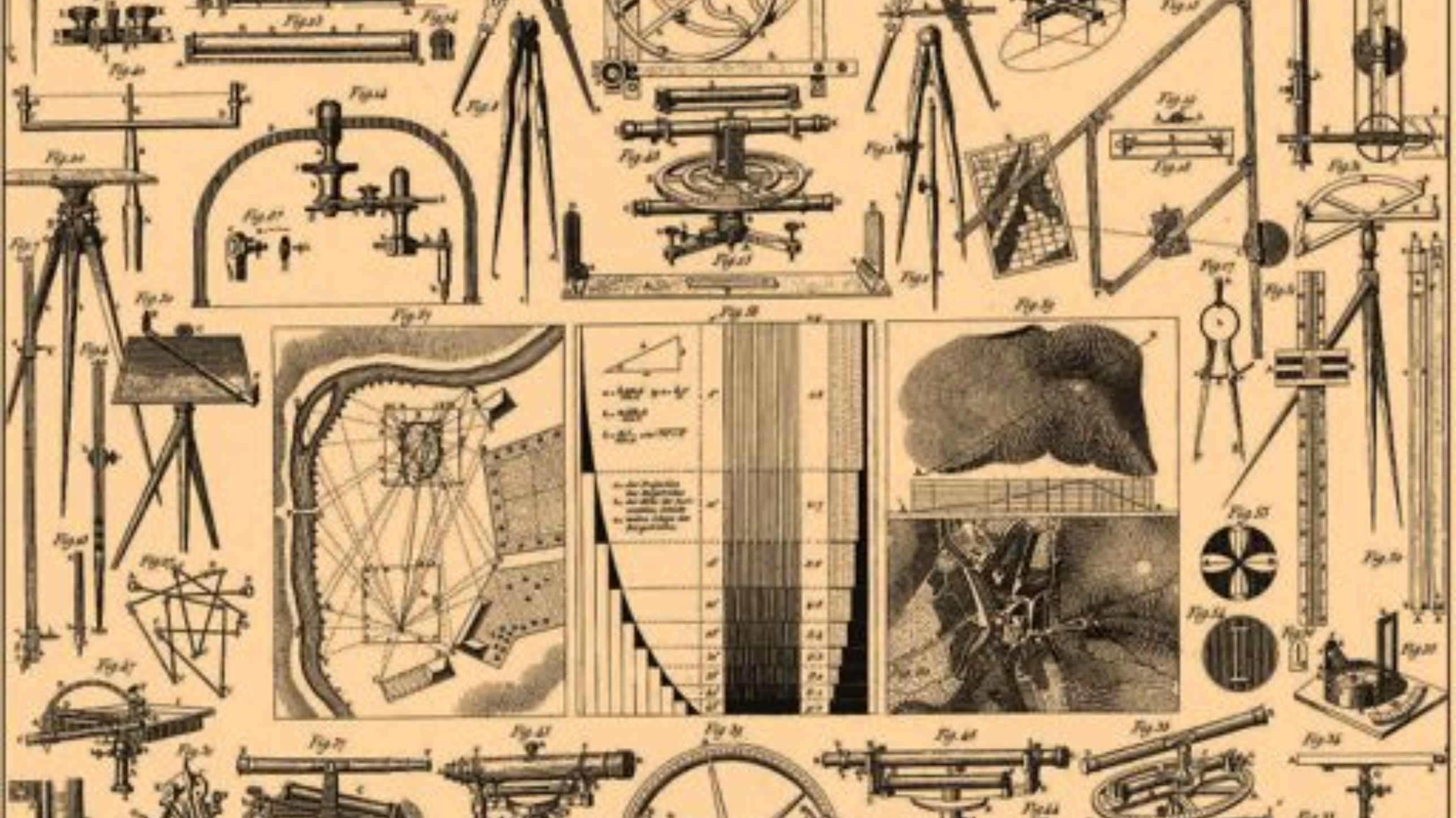
```
function delegate (receiver, metaobject, methods) {
  if (methods == null) {
    methods = Object.keys(metaobject).filter(function (methodName) {
      return typeof(metaobject[methodName]) == 'function';
    });
  }
  methods.forEach(function (methodName) {
    receiver[methodName] = function () {
      return metaobject[methodName].apply(receiver, arguments);
    };
  });

  return receiver;
};
```

COULD THERE BE ANOTHER WAY TO DELEGATE TO A METAOBJECT?

YES.

```
Object.create(metaobject);
```

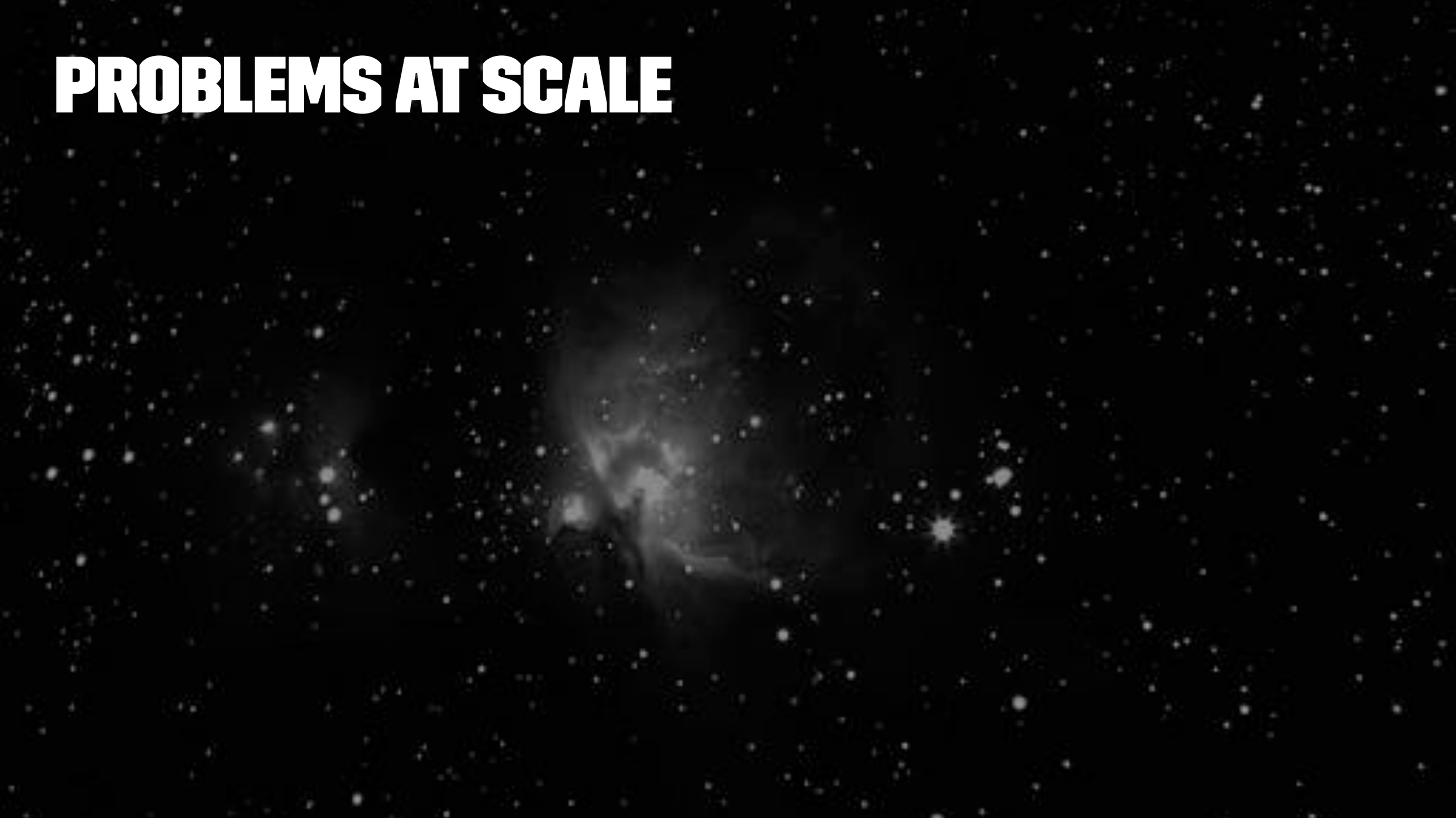

INHERITENCE IS
DELEGATION THROUGH
PROTOTYPES



NOW THAT WE'VE CATEGORIZED OUR BASIC TOOLS...

WE NEED TO THINK AT SCALE

PROBLEMS AT SCALE



PROBLEMS AT SCALE
- OBJECT COUPLING

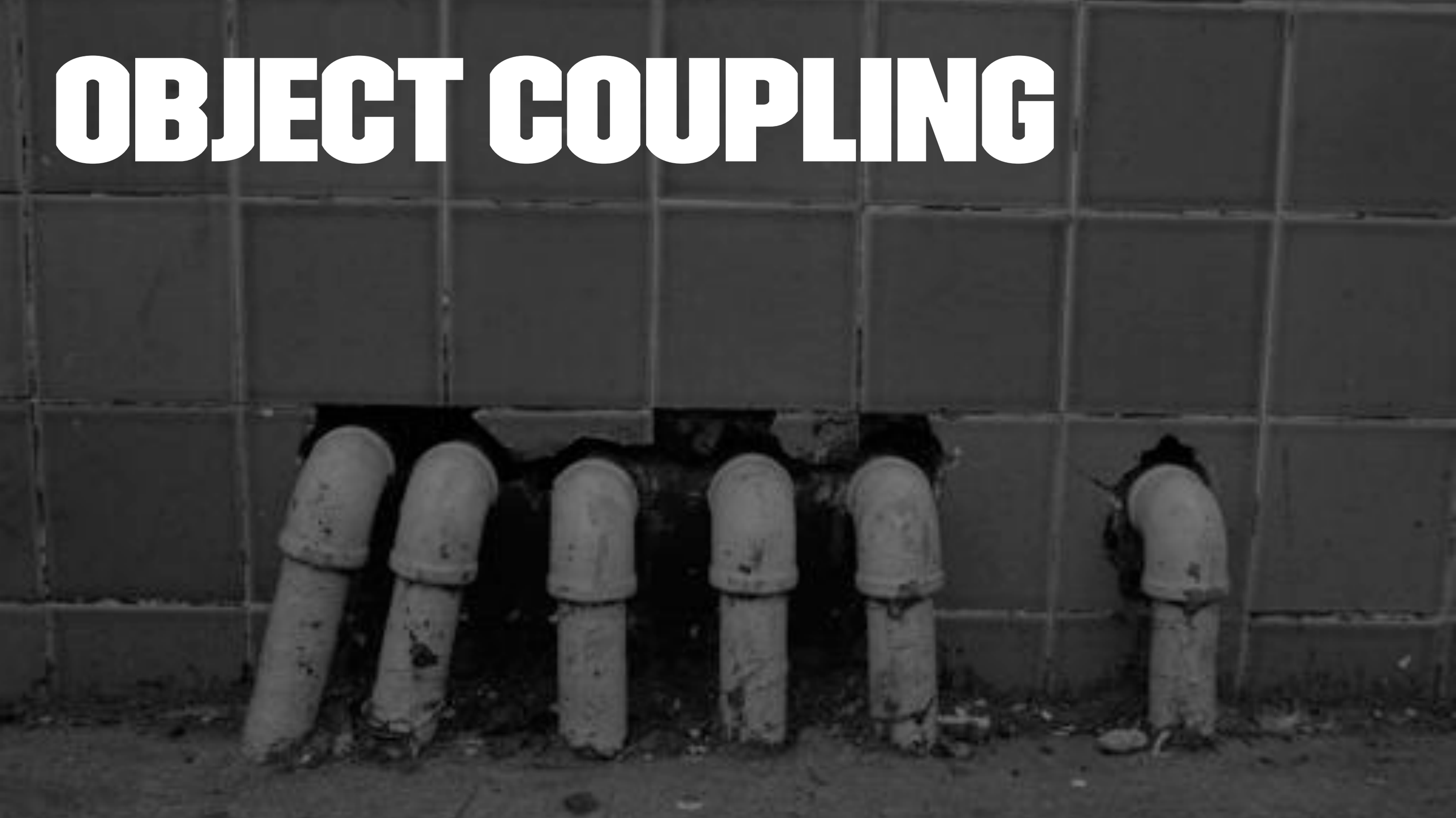
PROBLEMS AT SCALE

- OBJECT COUPLING
- INFLEXIBILITY

PROBLEMS AT SCALE

- OBJECT COUPLING
- INFLEXIBILITY
- METAOBJECT COUPLING

OBJECT COUPLING



“OOP TO ME MEANS ONLY MESSAGING, LOCAL RETENTION & PROTECTION & HIDING OF STATE-PROCESS*”

Dr. Alan Kay

WE HAVE A PATTERN FOR PRIVATE STATE: FORWARDING

```
function proxy (baseObject, optionalPrototype) {
  var proxyObject = Object.create(optionalPrototype || null),
      methodName;
  for (methodName in baseObject) {
    if (typeof(baseObject[methodName]) ===  'function') {
      (function (methodName) {
        proxyObject[methodName] = function () {
          var result = baseObject[methodName].apply(
                       baseObject,
                       arguments
                     );
          return (result === baseObject)
                 ? proxyObject
                 : result;
        }
      })(methodName);
    }
  }
  return proxyObject;
}
```

```
var stack = {
  array: [],
  index: -1,
  push: function (value) {
    return this.array[this.index += 1] = value;
  },
  pop: function () {
    var value = this.array[this.index];
    this.array[this.index] = void 0;
    if (this.index >= 0) {
      this.index -= 1;
    }
    return value;
  },
  isEmpty: function () {
    return this.index < 0;
  }
};
```

```
var stackProxy = proxy(stack);

stackProxy.push('first');

stackProxy
  //=>
    { push: [Function],
      pop: [Function],
      isEmpty: [Function] }

stackProxy.pop();
  //=> 'first'
```

INFLEXIBILITY

PROTOTYPE INHERITENCE IS ONE TO MANY

MIXINS ARE MANY TO MANY

PERSON

```
var Person = {
  fullName: function () {
    return this.firstName + " " + this.lastName;
  },
  rename: function (first, last) {
    this.firstName = first;
    this.lastName = last;
    return this;
  }
};
```

HAS-CAREER

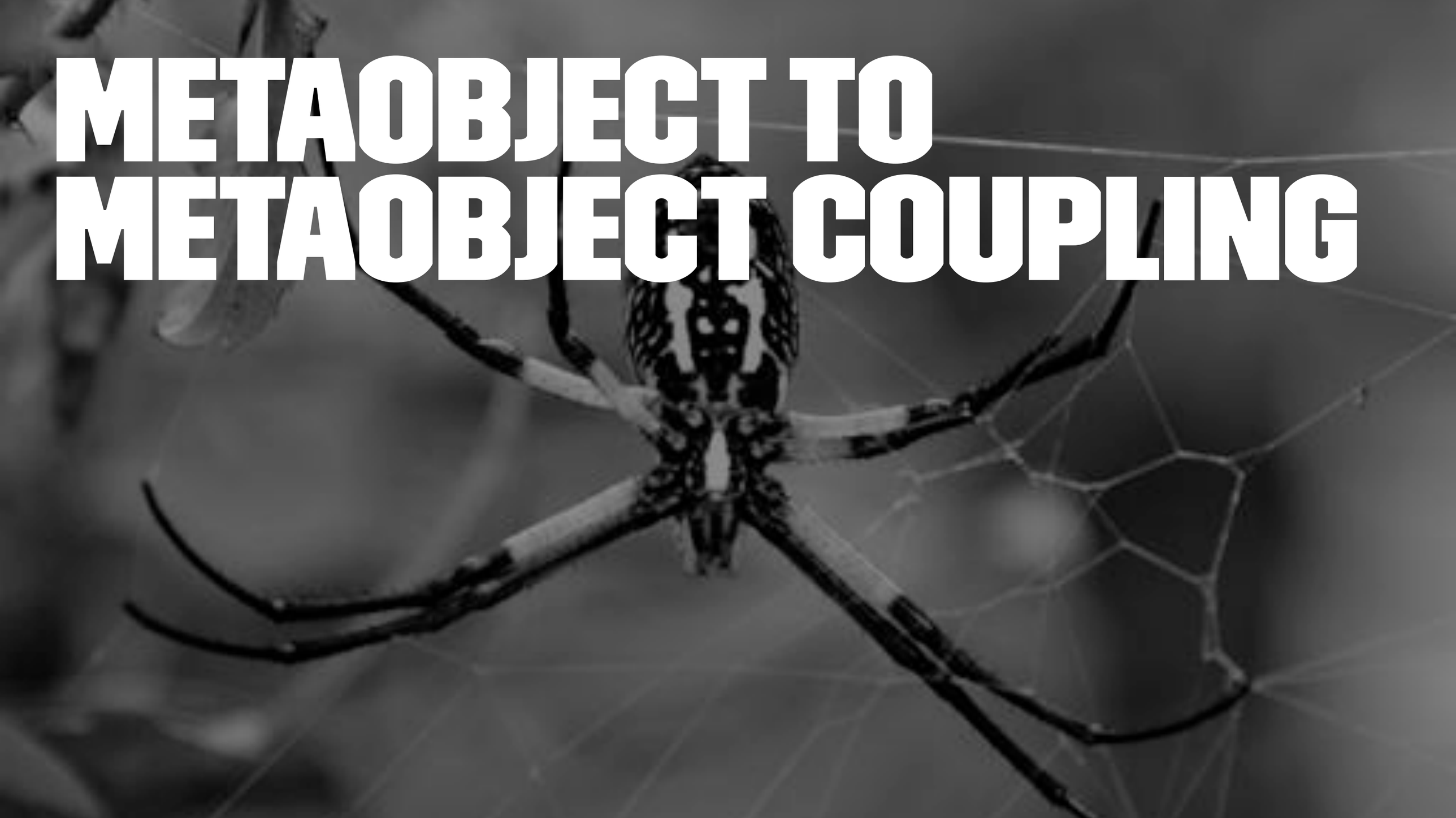
```
var HasCareer = {
  career: function () {
    return this.chosenCareer;
  },
  setCareer: function (career) {
    this.chosenCareer = career;
    return this;
  }
};
```

MODERN CAREERIST

```
var Careerist = extend(
    Object.create(null),
    Person,
    HasCareer
);

var sam = Object.create(Careerist);
```

TRADITIONAL CAREERIST

```
function Careerist () {}

Careerist.prototype = extend(
    Object.create(null),
    Person,
    HasCareer
);

var sam = new Careerist();
```

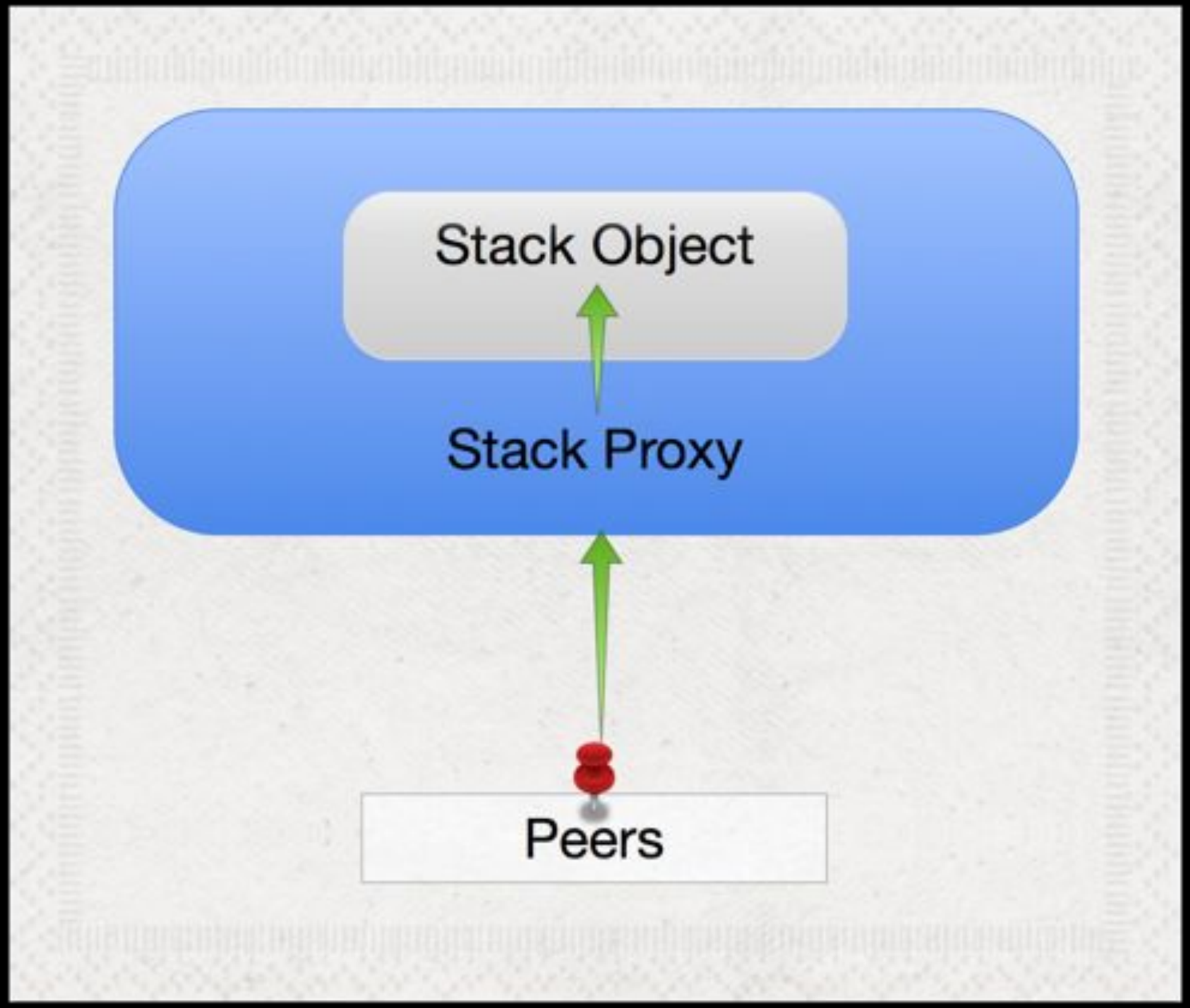

METAOBJECT TO
METAOBJECT COUPLING


ENCAPSULATION FOR
METAOBJECTS

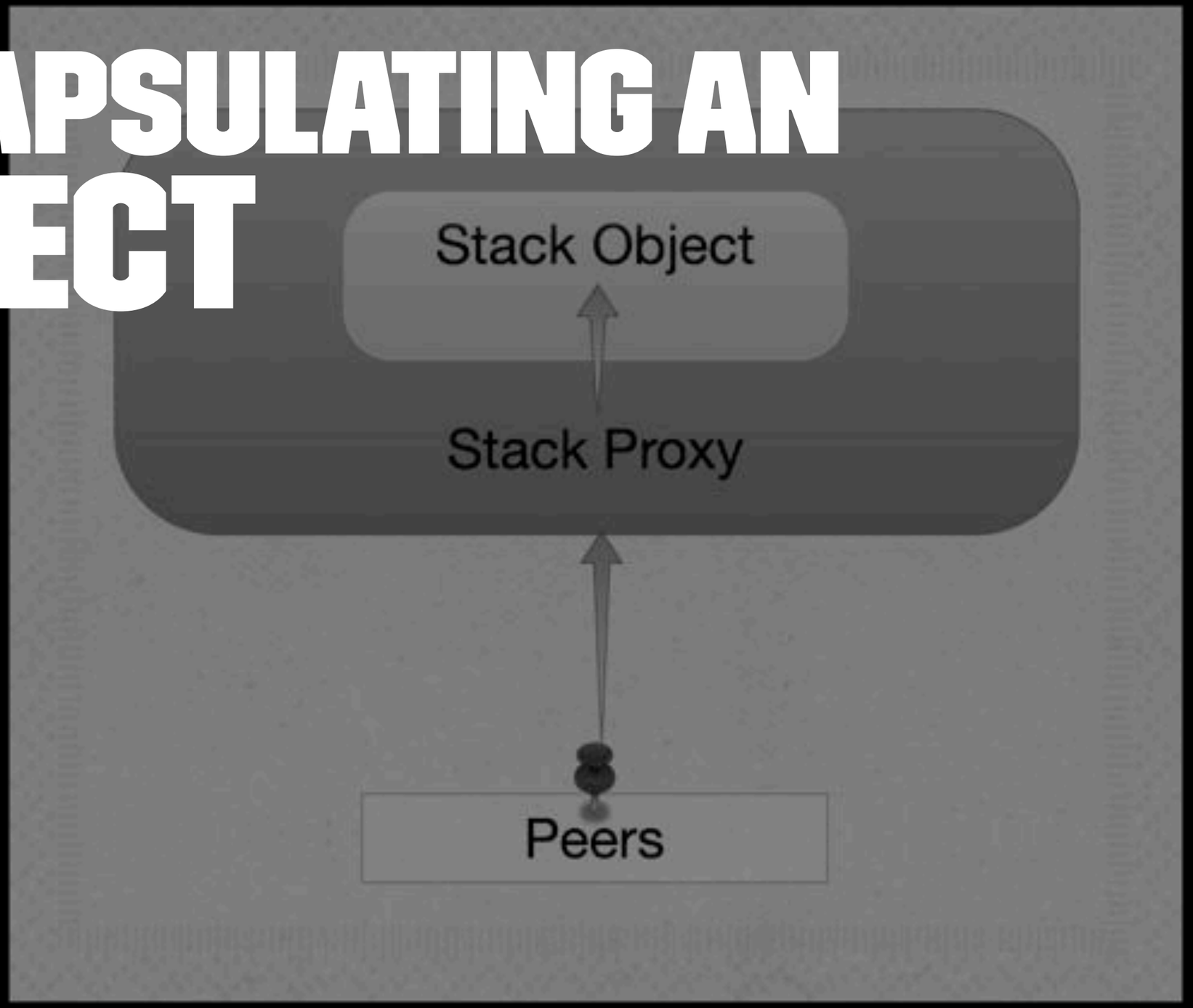
OPEN RECURSION CONSIDERED HARMFUL

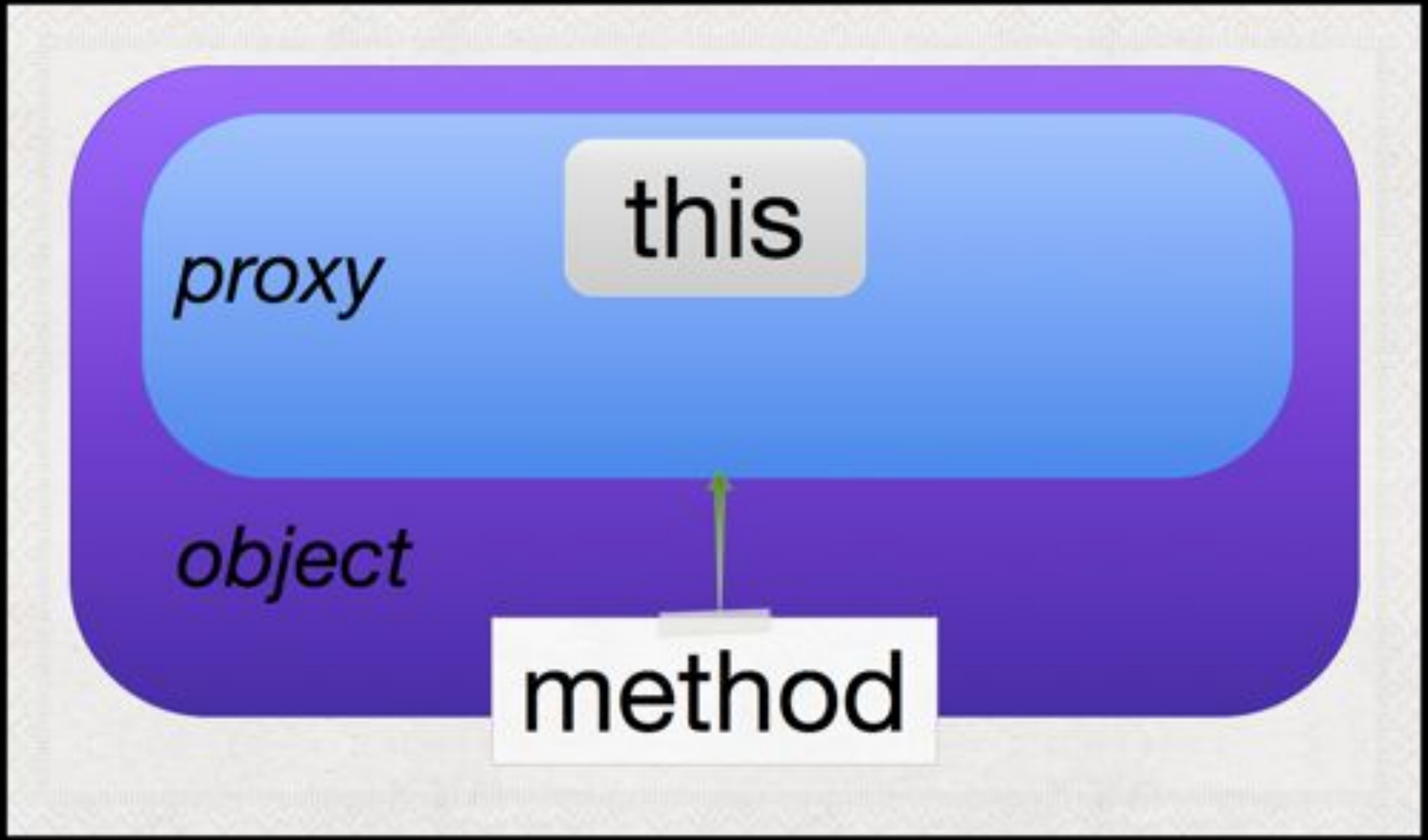
The fragile base class problem is a fundamental architectural problem of object-oriented programming systems where base classes (superclasses) are considered "fragile" because seemingly safe modifications to a base class, when inherited by the derived classes, may cause the derived classes to malfunction.


ENCAPSULATION FOR
METAOBJECTS

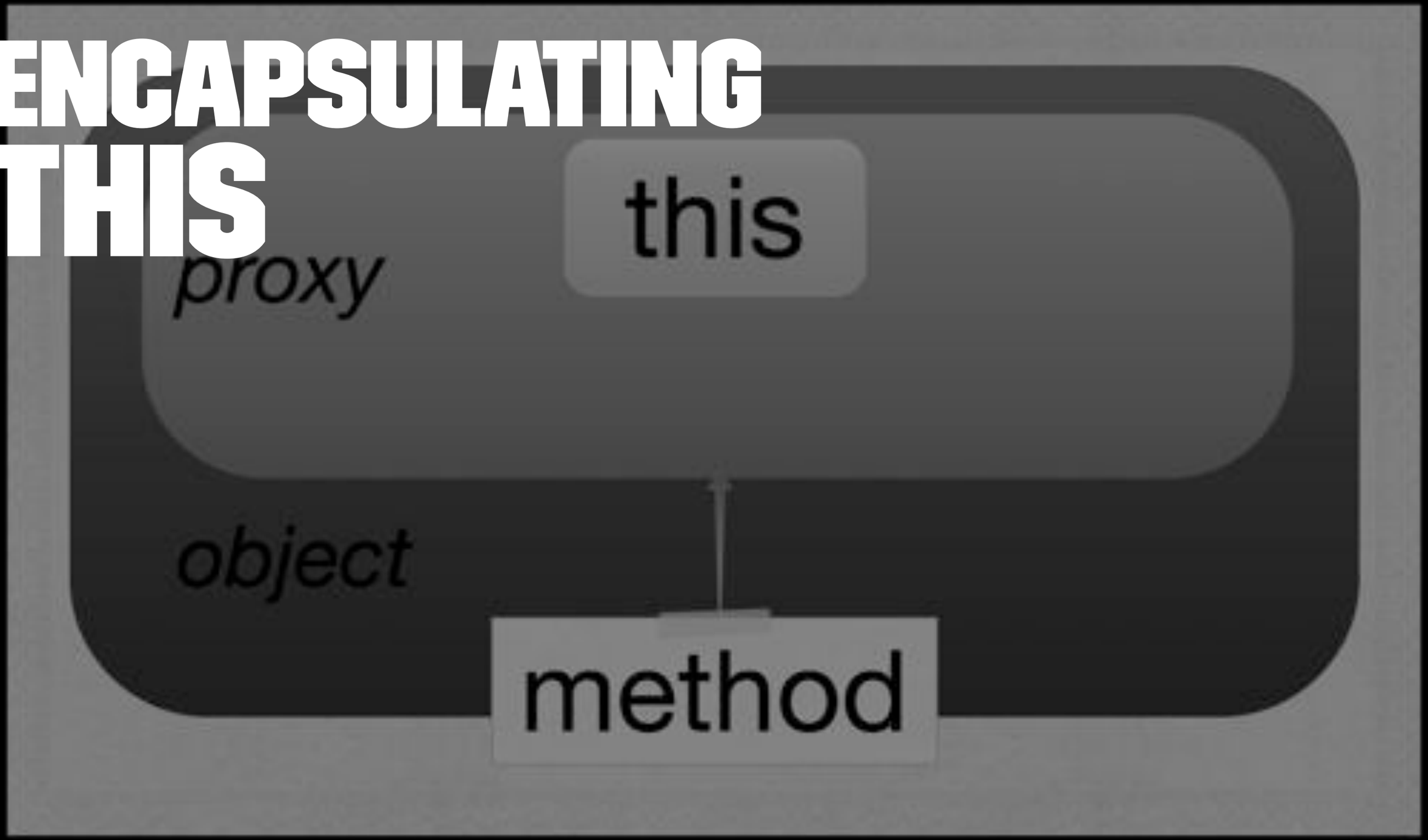

Stack Object
Stack Proxy
Peers

ENCAPSULATING AN OBJECT




proxy
this
object
method

ENCAPSULATING THIS



```
var number = 0;

function encapsulate (behaviour) {
  var safekeepingName = "__" + ++number + "__",
      encapsulatedObject = {};

  function createContext (methodReceiver) {
    return proxy(methodReceiver);
  }

  function getContext (methodReceiver) {
    var context = methodReceiver[safekeepingName];
    if (context == null) {
      context = createContext(methodReceiver);
      Object.defineProperty(methodReceiver, safekeepingName, {
        enumerable: false,
        writable: false,
        value: context
      });
    }
    return context;
  }

  Object.keys(behaviour).forEach(function (methodName) {
    var methodBody = behaviour[methodName];

    encapsulatedObject[methodName] = function () {
      var context = getContext(this),
          result = description[methodName].apply(context, arguments);
      return (result === context) ? this : result;
    };
  });

  return encapsulatedObject;
}
```

ENCAPSULATION LACKS A SENSE OF SELF

```
function createContext (methodReceiver) {
  return Object.defineProperty(
    proxy(methodReceiver),
    'self',
    { writable: false, enumerable: false, value: methodReceiver }
  );
}
```

PRIVATE METHODS

```
var MultiTalented = encapsulate({
  _englishList: function (list) {
    var butLast = list.slice(0, list.length - 1),
        last = list[list.length - 1];
    return butLast.length > 0
           ? [butLast.join(', '), last].join(' and ')
           : last;
  },
  initialize: function () {
    this._careers = [];
    return this;
  },
  addCareer: function (career) {
    this._careers.push(career);
    return this;
  },
  careers: function () {
    return this._englishList(this._careers);
  }
});
```

PRIVATE METHODS ARE A KEY ENCAPSULATION IDEA

IF ONLY WE KNEW WHICH METHODS WERE PRIVATE...

```
function createContext (methodReceiver) {
    var innerProxy = proxy(methodReceiver);

    privateMethods.forEach(function (methodName) {
        innerProxy[methodName] = behaviour[methodName];
    });
  return Object.defineProperty(
    innerProxy,
    'self',
    { writable: false, enumerable: false, value: methodReceiver }
  );
}
```

```
function encapsulate (behaviour) {
  var privateMethods = methods.filter(function (methodName) {
          return methodName[0] === '_';
        }),
      publicMethods = methods.filter(function (methodName) {
          return methodName[0] !== '_';
        });

    // ...

  return publicMethods.reduce(function (acc, methodName) {
    var methodBody = behaviour[methodName];

    acc[methodName] = function () {
      var context = getContext(this),
          result = behaviour[methodName].apply(context, arguments);
      return (result === context) ? this : result;
    };
    return acc;
  }, {});
}
```

ENCAPSULATION'S BEARD IS HALF CONSTRUCTED

HAS-NAME IS INDEPENDENT

```
var HasName = encapsulate({
  name: function () {
    return this.name;
  },
  setName: function (name) {
    this.name = name;
    return this;
  }
});
```

HAS-CAREER IS INDEPENDENT

```
var HasCareer = encapsulate({
  career: function () {
    return this.name;
  },
  setCareer: function (name) {
    this.name = name;
    return this;
  }
});
```

IS-DELF DESCRIBING HAS DEPENDENCIES

```
var IsSelfDescribing = encapsulate({
  description: function () {
    return this.name() + ' is a ' + this.career();
  }
});
```

WE CAN
NAME OUR DEPENDENCIES

```
var IsSelfDescribing = encapsulate({
  name: undefined,
  career: undefined,

  description: function () {
    return this.name() + ' is a ' + this.career();
  }
});
```

METHODS-OF-TYPE

```
function methodsOfType (behaviour, type) {
  var methods = [],
      methodName;

  for (methodName in behaviour) {
    if (typeof(behaviour[methodName]) === type) {
      methods.push(methodName);
    }
  };
  return methods;
}
```

IDENTIFYING DEPENDENCIES

```
function encapsulate (behaviour) {
  var safekeepingName = "__" + ++number + "__",
      methods = Object.keys(behaviour).filter(function (methodName) {
          return typeof behaviour[methodName] === 'function';
        }),
      privateMethods = methods.filter(function (methodName) {
          return methodName[0] === '_';
        }),
      publicMethods = methods.filter(function (methodName) {
          return methodName[0] !== '_';
        }),
            dependencies = Object.keys(behaviour).filter(function (methodName) {
          return typeof behaviour[methodName] === 'undefined';
        });
```

PARTIAL PROXIES

```
function partialProxy (baseObject, methods, proxyPrototype) {
  var proxyObject = Object.create(proxyPrototype || null);

  methods.forEach(function (methodName) {
    proxyObject[methodName] = function () {
      var result = baseObject[methodName].apply(baseObject, arguments);
      return (result === baseObject)
             ? proxyObject
             : result;
    }
  });

  return proxyObject;
}
```

USING PARTIAL PROXIES

```
function createContext (methodReceiver) {
    var innerProxy = partialProxy(
                        methodReceiver,
                        publicMethods.concat(dependencies)
                    );

    privateMethods.forEach(function (methodName) {
        innerProxy[methodName] = behaviour[methodName];
    });
  return Object.defineProperty(
    innerProxy,
    'self',
    { writable: false, enumerable: false, value: methodReceiver }
  );
}
```

ENCAPSULATION'S KEY FEATURES

1. The Receiver is encapsulated in an unenumerated property.
2. Separation of context and self.
3. Private methods.
4. Named Dependencies

COMPOSING METAOBJECTS

```
var SingsSongs = encapsulate({
  _songs: null,

  initialize: function () {
    this._songs = [];
    return this;
  },
  addSong: function (name) {
    this._songs.push(name);
    return this;
  },
  songs: function () {
    return this._songs;
  }
});
```

```
var HasAwards = encapsulate({
  _awards: null,

  initialize: function () {
    this._awards = [];
    return this;
  },
  addAward: function (name) {
    this._awards.push(name);
    return this;
  },
  awards: function () {
    return this._awards;
  }
});
```

```
var AwardWinningSongwriter = extend(
                  Object.create(null),
                  SingsSongs,
                  HasAwards
              ),
    tracy = Object.create(AwardWinningSongwriter);

tracy.initialize();
tracy.songs()
  //=> undefined
```

LISKOV SUBSTITUTION PRINCIPLE

Substitutability is a principle in object-oriented programming. It states that, in a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S (i.e., objects of type S may substitute objects of type T) without altering any of the desirable properties of that program

```
function isUndefined (value) {
  return typeof value === 'undefined';
}

function isntUndefined (value) {
  return typeof value !== 'undefined';
}

function isFunction (value) {
  return typeof value === 'function';
}
```

```
function orderStrategy2 () {
  if (arguments.length === 1) {
    return arguments[0];
  }
  else {
    var fns = __slice.call(arguments, 0);

    return function composed () {
      var args    = arguments,
          context = this,
          values  = fns.map(function (fn) {
            return fn.apply(context, args);
          }).filter(isntUndefined);

      if (values.length > 0) {
        return values[values.length - 1];
      }
    }
  }
}
```

```
function propertiesToArrays (metaobjects) {
  return metaobjects.reduce(function (collected, metaobject) {
    var key;

    for (key in metaobject) {
      if (key in collected) {
        collected[key].push(metaobject[key]);
      }
      else collected[key] = [metaobject[key]]
    }
    return collected;
  }, {})
}
```

```
function resolveUndefineds (collected) {
  return Object.keys(collected).reduce(function (resolved, key) {
    var values = collected[key];

    if (values.every(isUndefined)) {
      resolved[key] = undefined;
    }
    else resolved[key] = values.filter(isntUndefined);

    return resolved;
  }, {});
}
```

```
function applyProtocol(seed, resolveds, protocol) {
  return Object.keys(resolveds).reduce( function (applied, key) {
    var value = resolveds[key];

    if (isUndefined(value)) {
      applied[key] = value;
    }
    else if (value.every(isFunction)) {
      applied[key] = protocol.apply(null, value);
    }
    else throw "Don't know what to do with " + value;

    return applied;
  }, seed);
}
```

```
function canBeMergedInto (object1, object2) {
  var prototype1 = Object.getPrototypeOf(object1),
      prototype2 = Object.getPrototypeOf(object2);

  if (prototype1 === null) return prototype2 === null;
  if (prototype2 === null) return true;
  if (prototype1 === prototype2) return true;

  return Object.prototype.isPrototypeOf.call(prototype2, prototype1);
}
```

```
var callLeft2 = (function () {
  if (typeof callLeft == 'function') {
    return callLeft;
  }
  else if (typeof allong === 'object' &&
           typeof allong.es === 'object' &&
           typeof allong.es.callLeft === 'function') {
    return allong.es.callLeft;
  }
  else {
    return function callLeft2 (fn, arg2) {
      return function callLeft2ed (arg1) {
        return fn.call(this, arg1, arg2);
      };
    };
  }
})();
```

```
function seedFor (objectList) {
  var seed = objectList[0] == null
             ? Object.create(null)
             : Object.create(
                 Object.getPrototypeOf(objectList[0])
               ),
      isCompatibleWithSeed = callLeft2(canBeMergedInto, seed);

  if (!objectList.every(isCompatibleWithSeed)) {
        throw 'incompatible prototypes';
    }
  return seed;
}
```

```
function composeMetaobjects () {
  var metaobjects = __slice.call(arguments, 0),
      arrays      = propertiesToArrays(metaobjects),
      resolved    = resolveUndefineds(arrays),
      seed        = seedFor(metaobjects),
      composed    = applyProtocol(seed, resolved, orderStrategy2);

  return composed;
}
```

COMPOSE-METAOBJECTS IN ACTION

```
var Songwriter = encapsulate({
  initialize: function () {
    this._songs = [];
    return this.self;
  },
  addSong: function (name) {
    this._songs.push(name);
    return this.self;
  },
  songs: function () {
    return this._songs;
  }
});
```

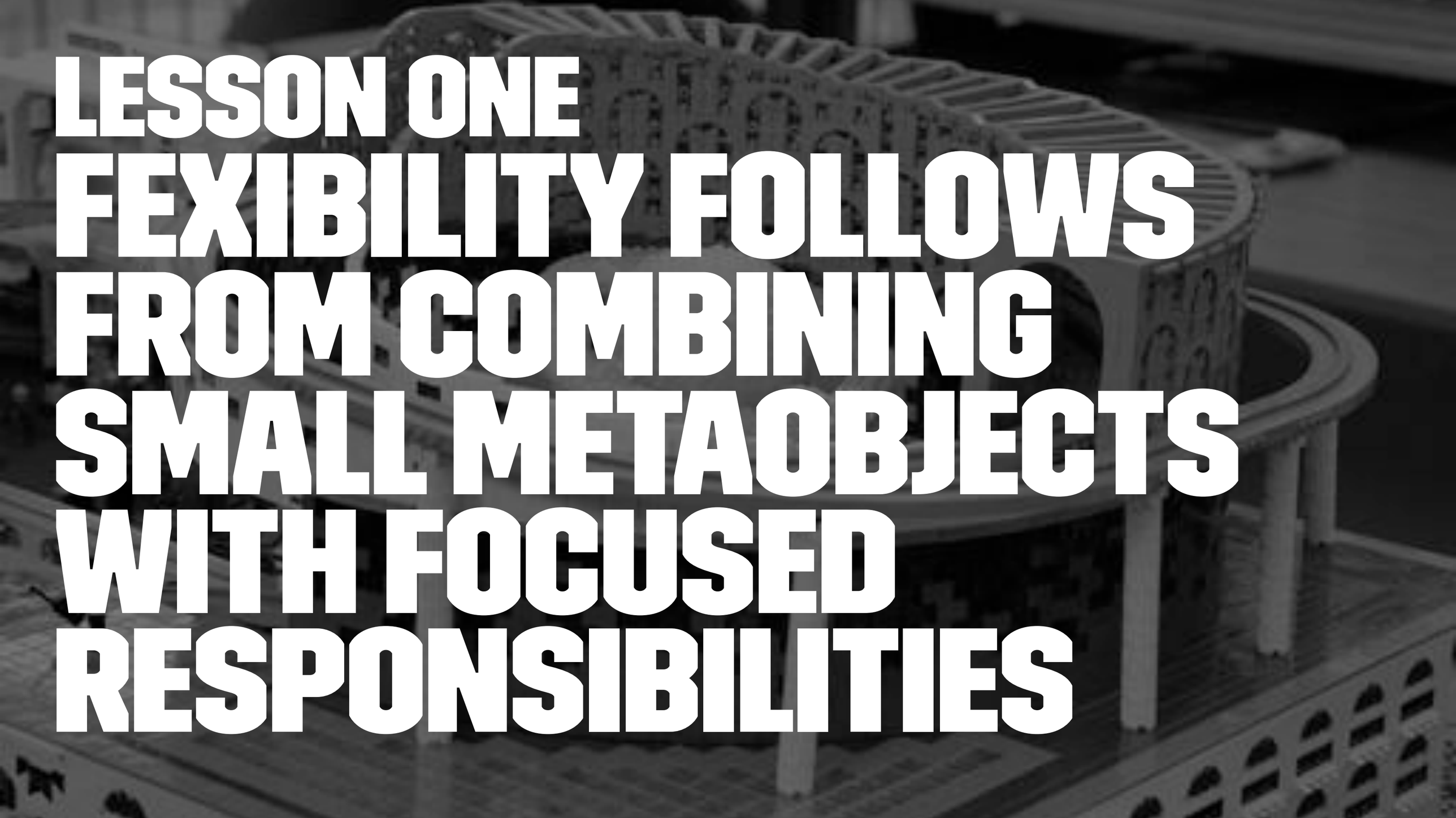
```
var Subscribable = encapsulate({
  initialize: function () {
    this._subscribers = [];
    return this.self;
  },
  subscribe: function (callback) {
    this._subscribers.push(callback);
  },
  unsubscribe: function (callback) {
    this._subscribers = this._subscribers.filter( function (subscriber) {
      return subscriber !== callback;
    });
  },
  subscribers: function () {
    return this._subscribers;
  },
  notify: function () {
    receiver = this;
    this._subscribers.forEach( function (subscriber) {
      subscriber.apply(receiver.self, arguments);
    });
  }
});
```

```
var SubscribableSongwriter = composeMetaobjects(
  Object.create(Songwriter),
  Subscribable,
  encapsulate({
    notify: undefined,
    addSong: function () { this.notify(); }
  })
);
```

```
var SongwriterView = {
  initialize: function (model, name) {
    this.model = model;
    this.name = name;
    this.model.subscribe(this.render.bind(this));
    return this;
  },
  _englishList: function (list) {
    var butLast = list.slice(0, list.length - 1),
        last = list[list.length - 1];
    return butLast.length > 0
           ? [butLast.join(', '), last].join(' and ')
           : last;
  },
  render: function () {
    var songList  = this.model.songs().length > 0
                  ? [" has written " + this._englishList(this.model.songs().map(function (song) {
                      return "'" + song + "'"; }))]
                  : [];

    console.log(this.name + songList);
    return this;
  }
};
```

```
var paulSimon = Object
                  .create(SubscribableSongwriter)
                                  .initialize(),
    paulView  = Object
                      .create(SongwriterView)
                                  .initialize(paulSimon, 'Paul Simon');

paulSimon.addSong('Cecilia')
  //=> Paul Simon has written 'Cecilia'
        {}

paulSimon.songs()
```

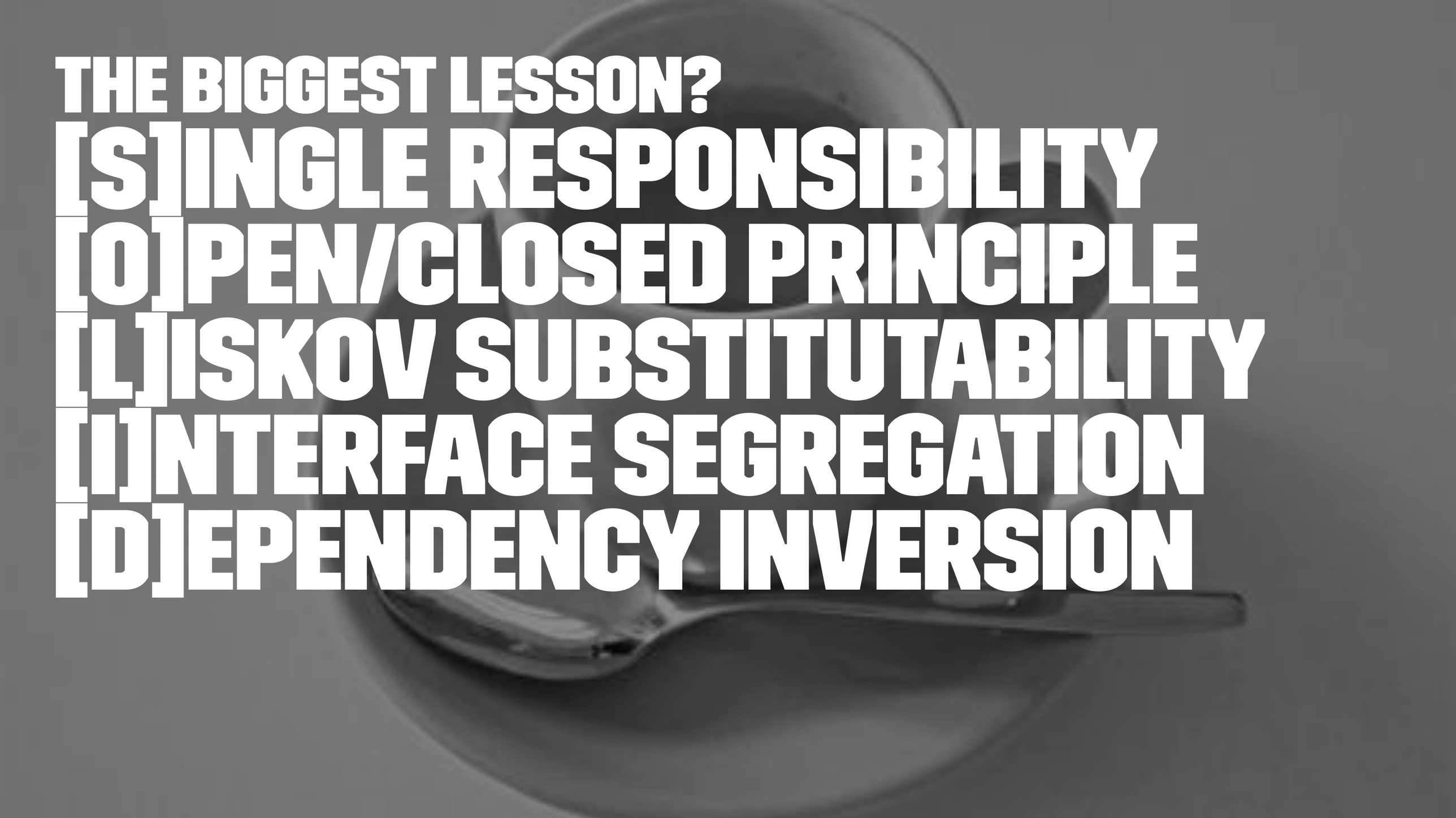
LESSONS


LESSON ONE
FEXIBILITY FOLLOWS
FROM COMBINING
SMALL METAOBJECTS
WITH FOCUSED
RESPONSIBILITIES

LESSON TWO
ENCAPSULATION REDUCES CONVENIENCE, BUT INCREASES FLEXIBILITY


ROW

LESSON THREE
CAREFUL COMPOSITION CREATES COHESION WITHOUT COUPLING



THE BIGGEST LESSON?

[S]INGLE RESPONSIBILITY
[O]PEN/CLOSED PRINCIPLE
[L]ISKOV SUBSTITUTABILITY
[I]NTERFACE SEGREGATION
[D]EPENDENCY INVERSION

REGINALD BRAITHWAITE

GITHUB, INC.
RAGANWALD.COM
@RAGANWALD

NDC Conference, Oslo, Norway,
June 5, 2014