Act III

Function the Ultimate

Function

Method

Class

Constructor

Module

`function` expression

- **`function`**
- optional name
- parameters
  - Wrapped in parens
  - Zero or more names
  - Separated by `,` (comma)
- body
  - Wrapped in curly braces
  - Zero or more statements

`function` expression

- Produces an instance of a function object.
- Function objects are first class.
  - May be passed as an argument to a function
  - May be returned from a function
  - May assigned to a variable
  - May be stored in an object or array
- Function objects inherit from **`Function.prototype`**.

`var` statement

- Declares and initializes variables within a function.
- Types are not specified.
- A variable declared anywhere within a function is visible everywhere within the function.

`var` statement

- It gets split into two parts:
  - The declaration part gets hoisted to the top of the function, initializing with **`undefined`**.
  - The initialization part turns into an ordinary assignment.

```
var myVar = 0, myOtherVar;
```

- Expands into

```
var myVar = undefined,
    myOtherVar = undefined;
    …
myVar = 0;
```

function statement

- **`function`**
- mandatory name
- parameters
  - Wrapped in parens
  - Zero or more names
  - Separated by `,` (comma)
- body
  - Wrapped in curly braces
  - Zero or more statements

`function` statement

- The `function` statement is a short-hand for a `var` statement with a function value.

```
function foo() {}
```

expands to

```
var foo = function foo() {};
```

which further expands to

```
var foo = undefined;
foo = function foo() {};
```

The assignment of the function is also hoisted.

`function` expression
v
`function` statement

If the first token in a statement is `function`, then it is a function statement.

Scope

Block scope v function scope

Scope

- In JavaScript, {blocks} do not have scope.
- Only functions have scope.
- Variables defined in a function are not visible outside of the function. Don't do this:

```
function assure_positive(matrix, n) {
    for (var i = 0; i < n; i += 1) {
        var row = matrix[i];
        for (var i = 0; i < row.length;
                i += 1) {
            if (row[i] < 0) {
                throw new Error('Negative');
}   }   }   }
```

Declare all variables at the top of the function.

Declare all functions before you call them.

The language provides mechanisms that allow you to ignore this advice, but they are problematic.

Return statement

```
return expression;
```

or

```
return;
```

- If there is no *expression*, then the return value is **`undefined`**.
- Except for constructors, whose default return value is **`this`**.

Two pseudo parameters

`arguments`

`this`

`arguments`

- When a function is invoked, in addition to its parameters, it also gets a special parameter called **`arguments`**.
- It contains all of the arguments from the invocation.
- It is an array-like object.
- **`arguments.length`** is the number of arguments passed.
- Weird interaction with parameters.

Example

```
function sum() {
    var i,
        n = arguments.length,
        total = 0;
    for (i = 0; i < n; i += 1) {
        total += arguments[i];
    }
    return total;
}

var ten = sum(1, 2, 3, 4);
```

`this`

- The **`this`** parameter contains a reference to the object of invocation.
- **`this`** allows a method to know what object it is concerned with.
- **`this`** allows a single function object to service many objects.
- **`this`** is key to prototypal inheritance.

Invocation

- The （ ） suffix operator surrounding zero or more comma separated arguments.
- The arguments will be bound to parameters.

Invocation

- If a function is called with too many arguments, the extra arguments are ignored.
- If a function is called with too few arguments, the missing values will be **`undefined`**.
- There is no implicit type checking on the arguments.

Invocation

- There are four ways to call a function:
  - Function form
    - *functionObject*`(`*arguments*`)`
  - Method form
    - *thisObject*`.`*methodName*`(`*arguments*`)`
    - *thisObject*`["`*methodName*`"](`*arguments*`)`
  - Constructor form
    - **`new`** *FunctionObject*`(`*arguments*`)`
  - Apply form
    - *functionObject*`.`**`apply`**`(`*thisObject*`,[`*arguments*`])`

Method form

thisObject`.`*methodName*`(`*arguments*`)`
thisObject`[`*methodName*`](`*arguments*`)`

- When a function is called in the method form, `this` is set to *thisObject*, the object containing the function.
- This allows methods to have a reference to the object of interest.

Function form

functionObject (*arguments*)

- When a function is called in the function form, **`this`** is set to the global object.
  - That is not very useful. (Fixed in ES5/Strict)
  - An inner function does not get access to the outer **`this`**.

```
var that = this;
```

Constructor form

`new` *FunctionValue* (*arguments*)

- When a function is called with the **`new`** operator, a new object is created and assigned to **`this`**.
- If there is not an explicit return value, then **`this`** will be returned.
- Used in the Pseudoclassical style.

Apply form

functionObject`.apply(`*thisObject, arguments*`)`
functionObject`.call(`*thisObject, argument…*`)`

- A function's **`apply`** or **`call`** method allows for calling the function, explicitly specifying *thisObject*.
- It can also take an array of parameters or a sequence of parameters.

```
Function.prototype.call = function (thisObject) {
    return this.apply(thisObject, Array
        .prototype.slice.apply(arguments, [1]));
};
```

`this`

- `this` is an bonus parameter. Its value depends on the calling form.
- `this` gives methods access to their objects.
- `this` is bound at invocation time.

Invocation form	`this`
function	the global object `undefined`
method	the object
constructor	the new object
apply	argument

Side Effects

Subroutine

call & return

aka sub, procedure, proc, func, function, lambda

Why are there subroutines?

- Code reuse
- Decomposition
- Modularity
- Expressiveness
- Higher Order

Recursion

When a function calls itself.

Quicksort

1. Divide the array into two groups, low and high.
2. Call Quicksort on each group containing more than one element.

Tennent's Principle of Correspondence

```
function factorial(n) {
    var result = 1;                    // result: variable
    while (n > 1) {
        result *= n;
        n -= 1;
    }
    return result;
}
```

```
function factorial(n) {
    return (function (result) { // result: parameter
        while (n > 1) {
            result *= n;
            n -= 1;
        }
        return result;
    }(1));
}
```

Tennent's Principle of Correspondence

expression

Any expression or statement can be wrapped in an immediately invoked function without changing meaning…

```
(function () {
    return expression;
}())
```

Except

`var` `function`

`break` `continue` `return`

`this` `arguments`

Closure

Lexical Scoping

Static Scoping

Closure

- The context of an inner function includes the scope of the outer function.
- An inner function enjoys that context even after the parent functions have returned.

- Function scope works like block scope.

Global

```
var names = ['zero', 'one', 'two',
        'three', 'four', 'five', 'six',
        'seven', 'eight', 'nine'];

var digit_name = function (n) {
    return names[n];
};



alert(digit_name(3));    // 'three'
```

Slow

```
var digit_name = function (n) {
    var names = ['zero', 'one', 'two',
        'three', 'four', 'five', 'six',
        'seven', 'eight', 'nine'];


    return names[n];
};



alert(digit_name(3));    // 'three'
```

Closure

```
var digit_name = (function () {
    var names = ['zero', 'one', 'two',
        'three', 'four', 'five', 'six',
        'seven', 'eight', 'nine'];

    return function (n) {
        return names[n];
    };

}());

alert(digit_name(3));    // 'three'
```

Global

```
var names = ['zero', 'one', 'two',
        'three', 'four', 'five', 'six',
        'seven', 'eight', 'nine'];

var digit_name = function (n) {
    return names[n];
};



alert(digit_name(3));    // 'three'
```

Immediate function returns a function

```
var names = ['zero', 'one', 'two',
        'three', 'four', 'five', 'six',
        'seven', 'eight', 'nine'];
var digit_name = (function () {
    return function (n) {
        return names[n];
    };
}());



alert(digit_name(3));    // 'three'
```

Closure

```
var digit_name = (function () {
    var names = ['zero', 'one', 'two',
        'three', 'four', 'five', 'six',
        'seven', 'eight', 'nine'];

    return function (n) {
        return names[n];
    };

}());

alert(digit_name(3));    // 'three'
```

Lazy (Don’t Do This)

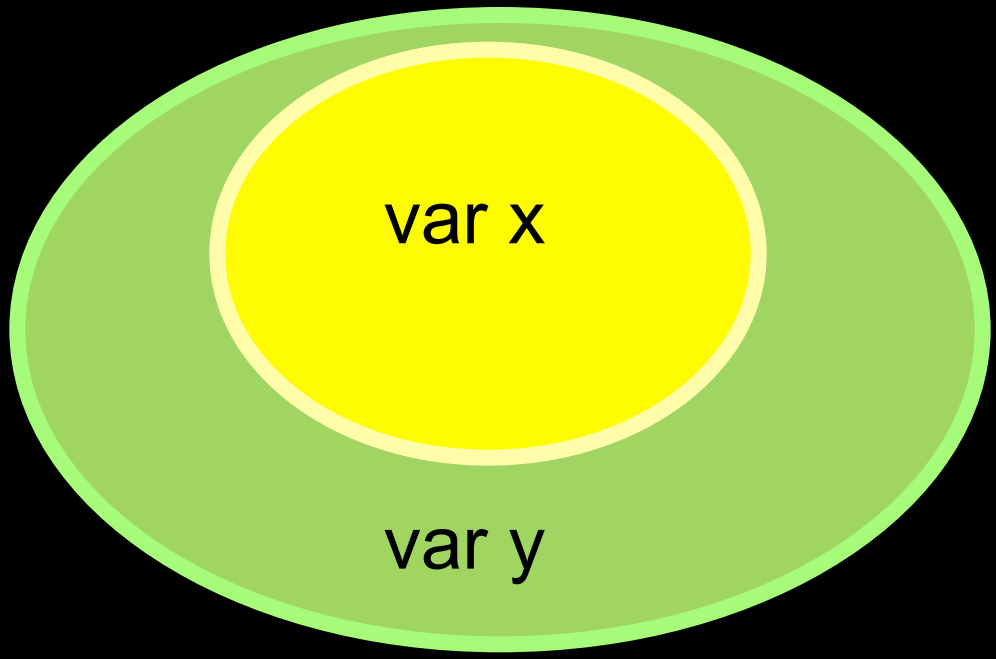
```
var digit_name = function (n) {
    var names = ['zero', 'one', 'two',
        'three', 'four', 'five', 'six',
        'seven', 'eight', 'nine'];

    digit_name = function (n) {
        return names[n];
    };
    return digit_name(n);
};

alert(digit_name(3));    // 'three'
```

Closure Conditional

```
var digit_name = (function () {
    var names;
    return function (n) {
        if (!names) {
            names = ['zero', 'one', 'two',
                'three', 'four', 'five', 'six',
                'seven', 'eight', 'nine'];
        }
        return names[n];
    };
}());
alert(digit_name(3));    // 'three'
```

Scope and Sets

```
(function outer() {
    var x;
// The inner function cannot see y
    return function inner(n) {
// The outer function can see x
        var y = x;
    };
}());
```



```
function fade(id) {
    var dom = document.getElementById(id),
        level = 1;
    function step() {
        var h = level.toString(16);
        dom.style.backgroundColor =
            '#FFFF' + h + h;
        if (level < 15) {
            level += 1;
            setTimeout(step, 100);
        }
    }
    setTimeout(step, 100);
}
```

`later` method

- The `later` method causes a method on the object to be invoked in the future.

```
my_object.later(1000, "erase", true);


arguments.slice(2)

Array.prototype.slice.apply(arguments, [2]);
```

`later` method

```
if (typeof Object.prototype.later !== 'function') }
    Object.prototype.later = function (msec, method) {
        var that = this,
            args = Array.prototype.slice
                    .apply(arguments, [2]);
        if (typeof method === 'string') {
            method = that[method];
        }
        setTimeout(function () {
            method.apply(that, args);
        }, msec);
        return that;    // Cascade
    };
}
```

Sealer/Unsealer

```
function make_sealer() {
    var boxes = [], values = [];

    return {
        sealer: function (value) {
            var i = boxes.length,
                box = {};
            boxes[i] = box;
            values[i] = value;
            return box;
        },
        unsealer: function (box) {
            return values[boxes.indexOf(box)];
        };
    };
}
```

statusHolder

```
var statusHolder = (function () {
    var status, subscribers = [];
    return {
        getStatus: function () {
            return status;
        },
        addListener: function (func) {
            if (typeof func !== 'function') {
                throw new TypeError('Expected a function.');
            }
            subscribers.push(func);
        },
        setStatus: function (newStatus) {
            var func, i, n = subscribers.length;
            status = newStatus;
            for (i = 0; i < n; i += 1) {
                func = subscribers[i];
                try {
                    func(newStatus);
                } catch (ignore) {};
            }
        }
    };
}());
```

Pseudoclassical

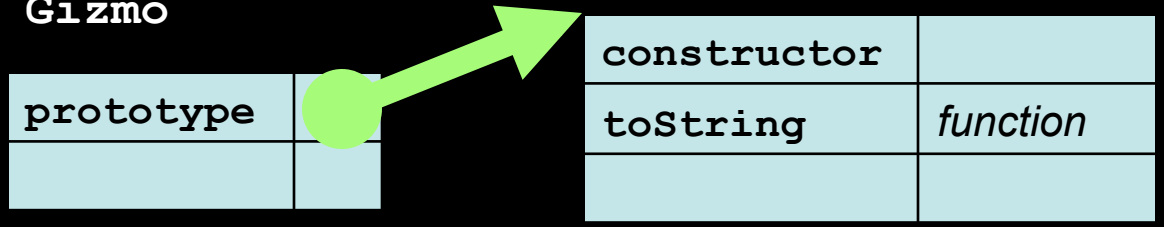
```
function Gizmo(id) {
    this.id = id;
}
Gizmo.prototype.toString = function () {
    return "gizmo " + this.id;
};
```

```
function Gizmo(id) {
    this.id = id;
}
Gizmo.prototype.toString = function () {
    return "gizmo " + this.id;
};
```

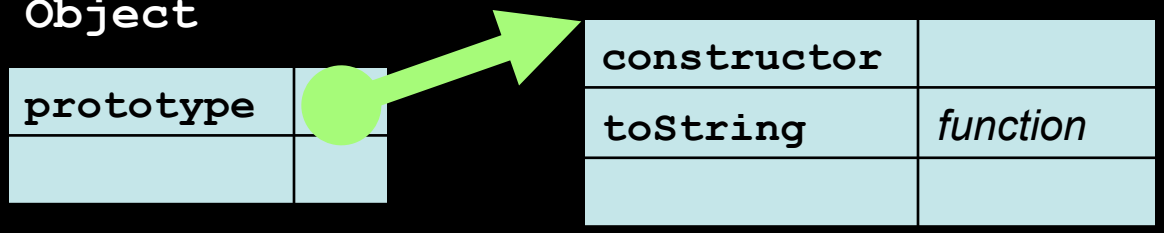
new Gizmo(*string*)

id	*string*

Gizmo



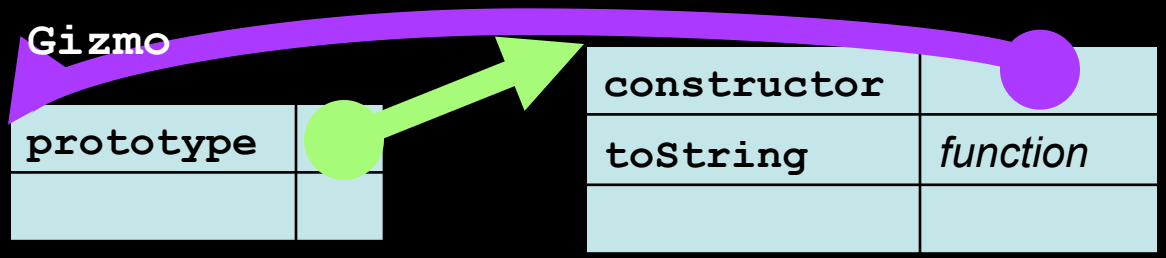
Object

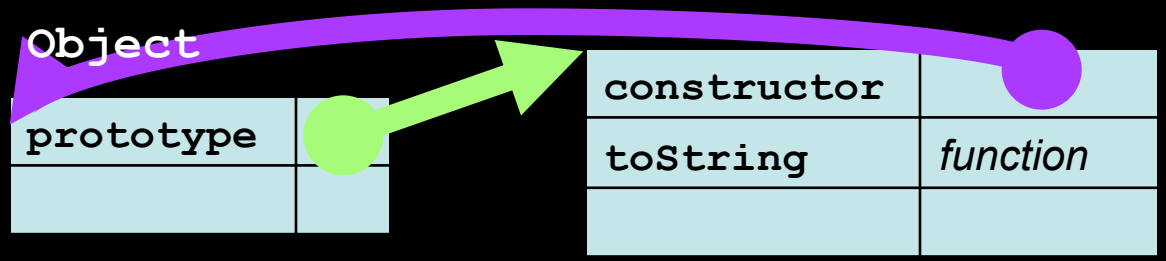


```
function Gizmo(id) {
    this.id = id;
}
Gizmo.prototype.toString = function () {
    return "gizmo " + this.id;
};
```

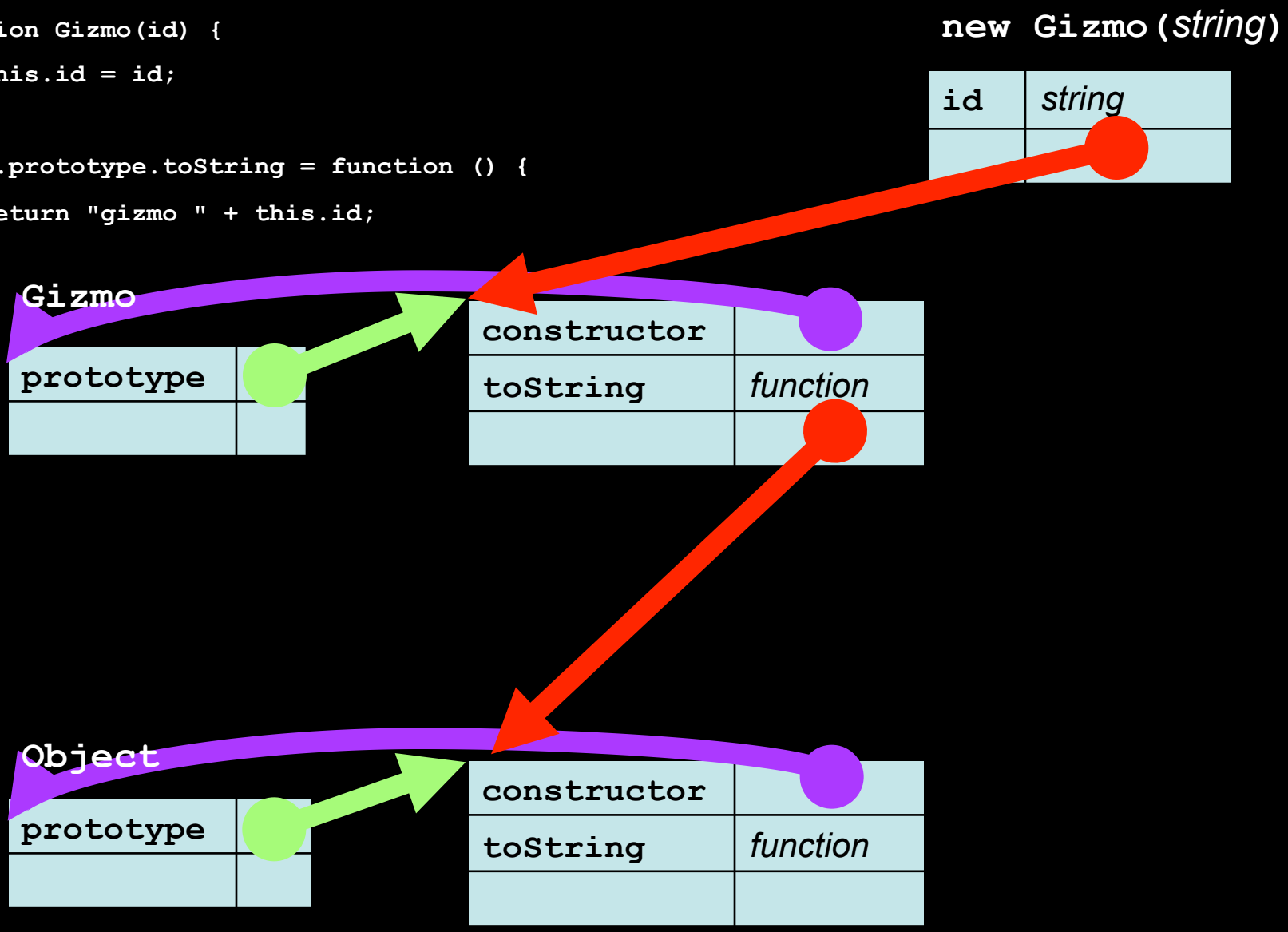
new Gizmo(*string*)

id	*string*





```
function Gizmo(id) {
    this.id = id;
}
Gizmo.prototype.toString = function () {
    return "gizmo " + this.id;
};
```



Pseudoclassical Inheritance

If we replace the original prototype object, then we can inherit another object's stuff.

```
function Hoozit(id) {
    this.id = id;
}
Hoozit.prototype = new Gizmo();
Hoozit.prototype.test = function (id) {
    return this.id === id;
};
```

```
function Hoozit(id) {
    this.id = id;
}
Hoozit.prototype = new Gizmo();
Hoozit.prototype.test = function (id) {
    return this.id === id;
};
```

new Hoozit(*string*)

id	*string*

Gizmo

prototype	

constructor	
toString	*function*

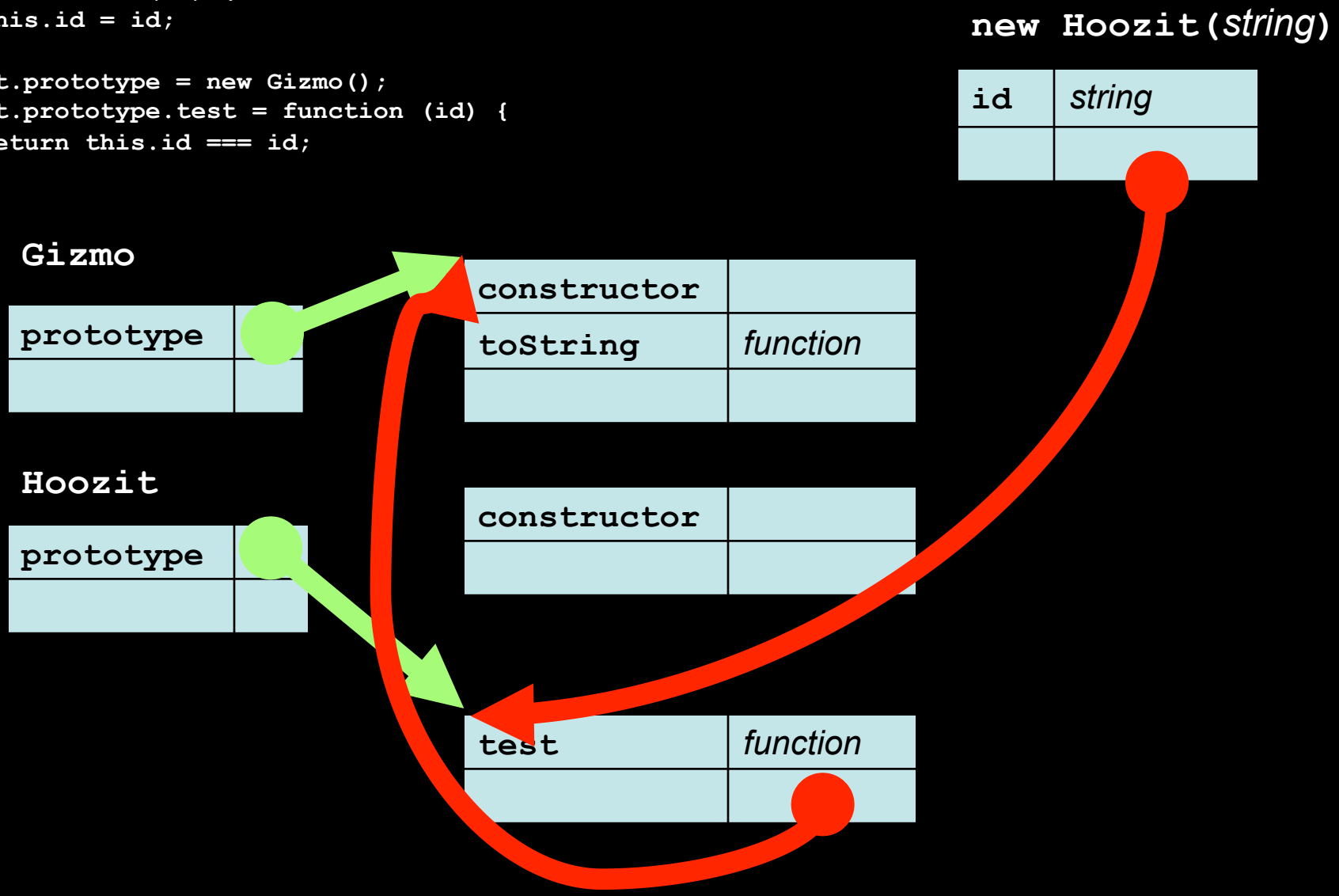
Hoozit

prototype	

constructor	

test	*function*

```
function Hoozit(id) {
    this.id = id;
}
Hoozit.prototype = new Gizmo();
Hoozit.prototype.test = function (id) {
    return this.id === id;
};
```



Pseudoclassical Inheritance

```
function Gizmo(id) {
    this.id = id;
}
Gizmo.prototype.toString = function () {
    return "gizmo " + this.id;
};


function Hoozit(id) {
    this.id = id;
}
Hoozit.prototype = new Gizmo();
Hoozit.prototype.test = function (id) {
    return this.id === id;
};
```

Prototypal Inheritance

```
var gizmo = new_constructor(Object, function (id) {
    this.id = id;
}, {
    toString: function () {
        return "gizmo " + this.id;
    }
});

var hoozit = new_constructor(gizmo, function (id) {
    this.id = id;
}, {
    test: function (id) {
        return this.id === id;
    }
});
```

```
function new_constructor(initializer, methods, extend) {
    var prototype = Object.create(typeof extend === 'function'
        ? extend.prototype
        : extend);
    if (methods) {
        methods.keys().forEach(function (key) {
            prototype[key] = methods[key];
        });
    }
    function constructor() {
        var that = Object.create(prototype);
        if (typeof initializer === 'function') {
            initializer.apply(that, arguments);
        }
        return that;
    }
    constructor.prototype = prototype;
    prototype.constructor = constructor;
    return constructor;
}
```

Function as module

```
var …
function …
function …
```

```
(function () {
    var …
    function …
    function …
}());
```

A Module Pattern

```
var singleton = (function () {
    var privateVariable;
    function privateFunction(x) {
        ...privateVariable...
    }
    return {
        firstMethod: function (a, b) {
            ...privateVariable...
        },
        secondMethod: function (c) {
            ...privateFunction()...
        }
    };
}());
```

A Module Pattern

```
(function () {
    var privateVariable;
    function privateFunction(x) {
        ...privateVariable...
    }
    GLOBAL.methodical = {
        firstMethod: function (a, b) {
            ...privateVariable...
        },
        secondMethod: function (c) {
            ...privateFunction()...
        }
    };
}());
```

Module pattern is easily transformed into a powerful constructor pattern.

Power Constructors

1. Make an object.
   - Object literal
   - `new`
   - `Object.create`
   - call another power constructor

Power Constructors

1. Make an object.
   - Object literal, `new`, `Object.create`, call another power constructor
2. Define some variables and functions.
   - These become private members.

Power Constructors

1. Make an object.
   - Object literal, `new`, `Object.create`, call another power constructor
2. Define some variables and functions.
   - These become private members.
3. Augment the object with privileged methods.

Power Constructors

1. Make an object.
   - Object literal, `new`, `Object.create`, call another power constructor
2. Define some variables and functions.
   - These become private members.
3. Augment the object with privileged methods.
4. Return the object.

Step One

```
function myPowerConstructor(x) {
    var that = otherMaker(x);
}
```

Step Two

```
function myPowerConstructor(x) {
    var that = otherMaker(x);
    var secret = f(x);
}
```

Step Three

```
function myPowerConstructor(x) {
    var that = otherMaker(x);
    var secret = f(x);
    that.priv = function () {
        ... secret x that ...
    };
}
```

Step Four

```
function myPowerConstructor(x) {
    var that = otherMaker(x);
    var secret = f(x);
    that.priv = function () {
        ... secret x that ...
    };
    return that;
}
```

Pseudoclassical Inheritance

```
function Gizmo(id) {
    this.id = id;
}
Gizmo.prototype.toString = function () {
    return "gizmo " + this.id;
};

function Hoozit(id) {
    this.id = id;
}
Hoozit.prototype = new Gizmo();
Hoozit.prototype.test = function (id) {
    return this.id === id;
};
```

Functional Inheritance

```
function gizmo(id) {
    return {
        id: id,
        toString: function () {
            return "gizmo " + this.id;
        }
    };
}

function hoozit(id) {
    var that = gizmo(id);
    that.test = function (testid) {
        return testid === this.id;
    };
    return that;
}
```

Privacy

```
function gizmo(id) {
    return {
        toString: function () {
            return "gizmo " + id;
        }
    };
}


function hoozit(id) {
    var that = gizmo(id);
    that.test = function (testid) {
        return testid === id;
    };
    return that;
}
```

Shared Secrets

```
function gizmo(id, secret) {
    secret = secret || {};
    secret.id = id;
    return {
        toString: function () {
            return "gizmo " + secret.id;
        };
    };
}

function hoozit(id) {
    var secret = {};    /*final*/
    var that = gizmo(id, secret);
    that.test = function (testid) {
        return testid === secret.id;
    };
    return that;
}
```

Super Methods

```
function hoozit(id) {
    var secret = {};
    var that = gizmo(id, secret);
    var super_toString = that.toString;
    that.test = function (testid) {
        return testid === secret.id;
    };
    that.toString = function () {
        return super_toString.apply(that);
    };
    return that;
}
```

```
function memoizer(memo, formula) {
    var recur = function (n) {
        var result = memo[n];
        if (typeof result !== 'number') {
            result = formula(recur, n);
            memo[n] = result;
        }
        return result;
    };
    return recur;
};

var factorial = memoizer([1, 1], function (recur, n) {
    return n * recur(n - 1);
});

var fibonacci = memoizer([0, 1], function (recur, n) {
    return recur(n - 1) + recur(n - 2);
});
```

Don’t make functions in a loop.

- It can be wasteful because a new function object is created on every iteration.
- It can be confusing because the new function closes over the loop’s variables, not over their current values.

Creating event handlers in a loop

```
for (var i ...) {
    div_id = divs[i].id;
    divs[i].onclick = function () {
        alert(div_id);
    };
}
```

```
var i;
function make_handler(div_id) {
    return function () {
        alert(div_id);
    }
}
for (i ...) {
    div_id = divs[i].id;
    divs[i].onclick = make_handler(div_id);
}
```

The Y Combinator

```
function y(le) {
    return (function (f) {
        return f(f);
    }(function (f) {
        return le(function (x) {
            return f(f)(x);
        });
    }));
}

var factorial = y(function (fac) {
    return function (n) {
        return n <= 2 ? n : n * fac(n - 1);
    };
});

var number120 = factorial(5);
```

JavaScript has good parts.

The Little Lisper

http://javascript.crockford.com/little.html

The Little Schemer

http://javascript.crockford.com/little.html

Later:

Episode IV
The Metamorphosis of Ajax

Currying

```
function curry(func) {
    var args = arguments.slice(1);
    return function () {
        return func.apply(null,
            args.concat(arguments.slice()));
    };
}

var inc = curry(function add(a, b) {
    return a + b;
}, 1);

alert(inc(6));    // 7
```