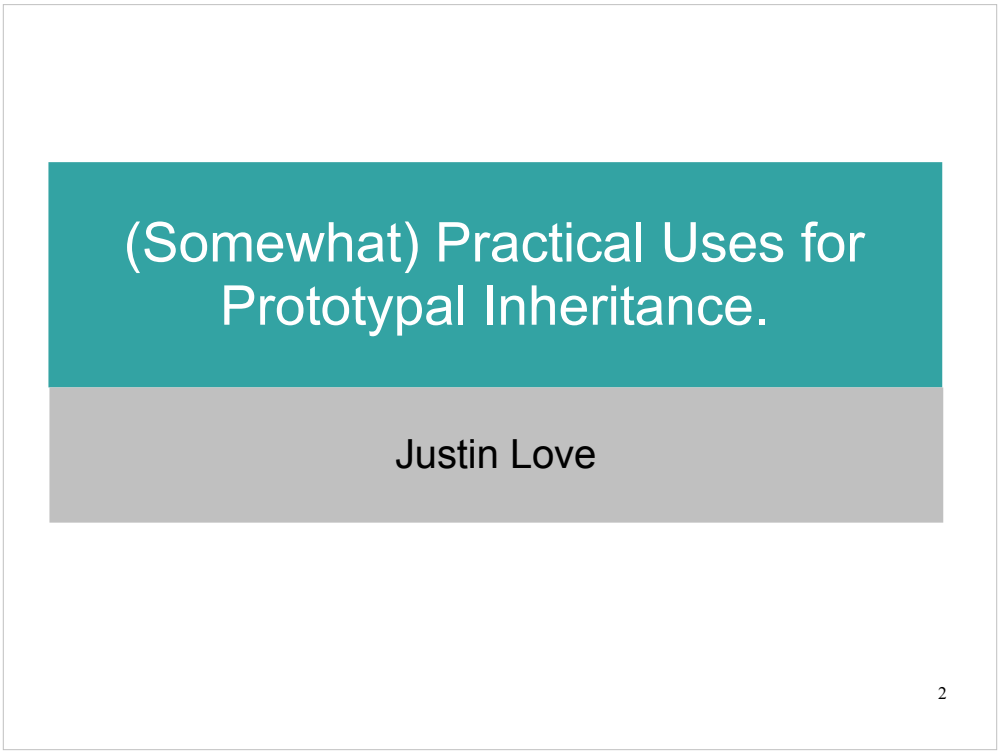
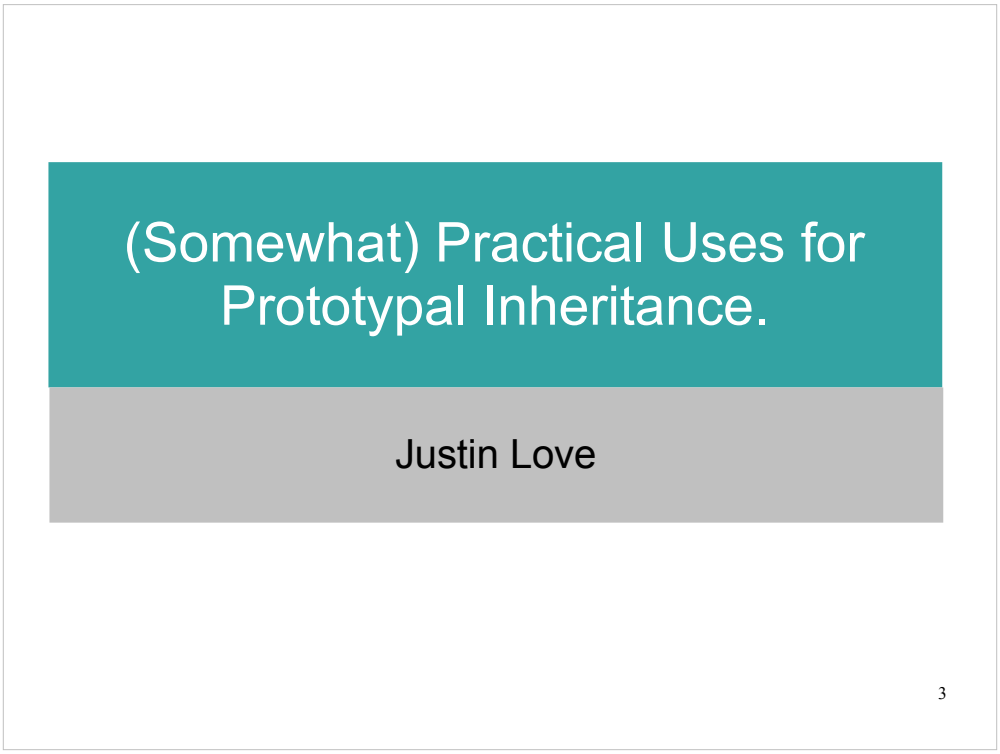


Howdy everybody.

My name is Justin Love. You might have noticed me lurking about the last few months. By day I work with embedded software and IT support, an occupation I'm in the process of disentangling myself from. By night I masquerade as a martial artist, code artist, and language geek.



I've heard or read about Javascript's peculiar object system several times.

It seems like the first thing everybody talks about is how to make it sit down and behave. But I've actually used prototypal inheritance in it's native form a few times and I thought I might be able to give you some ideas.

Overview

Concept Review

Example: tick marks

Example: parser state

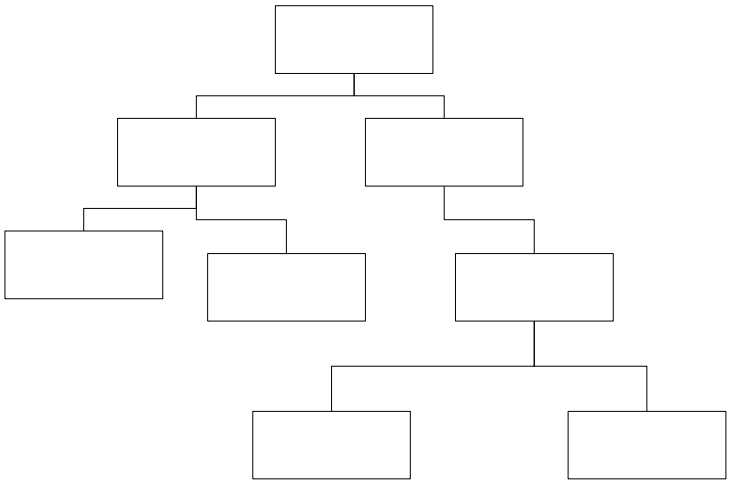
Some Patterns

First I'm going to go over some basic concepts and how we get a little closer to Javascript's true nature.

Then I'm going to cover some examples, mostly drawn from the project that me into Javascript in the first place.

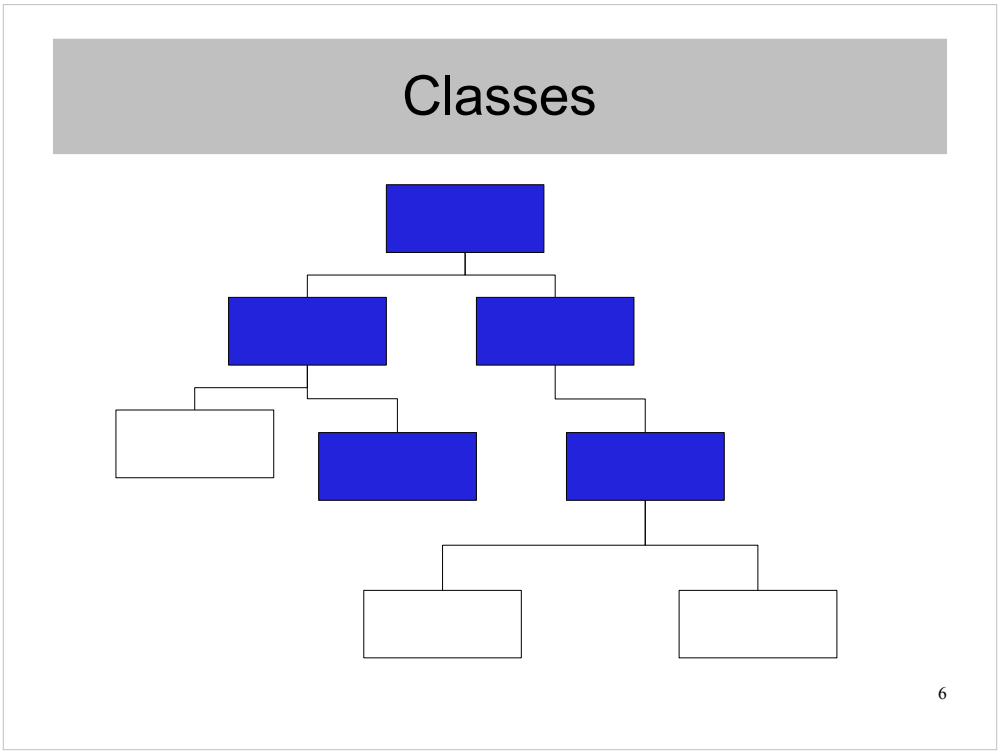
Finally, I'm going to review some common patterns that appear in and out of those examples.

Inheritance hierarchy

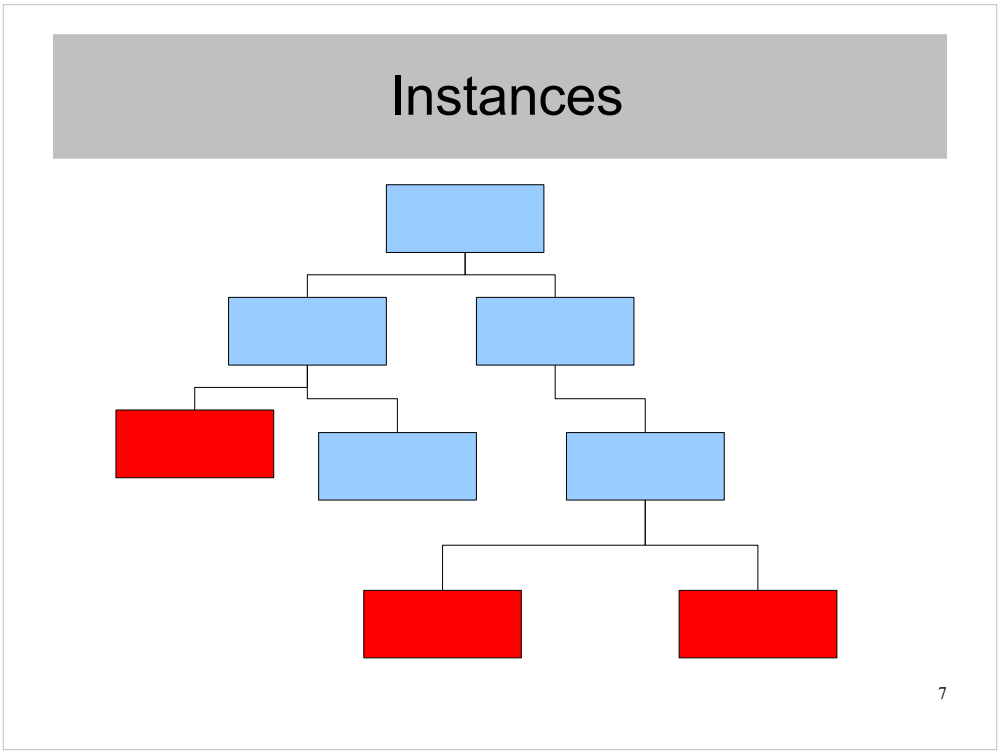


5

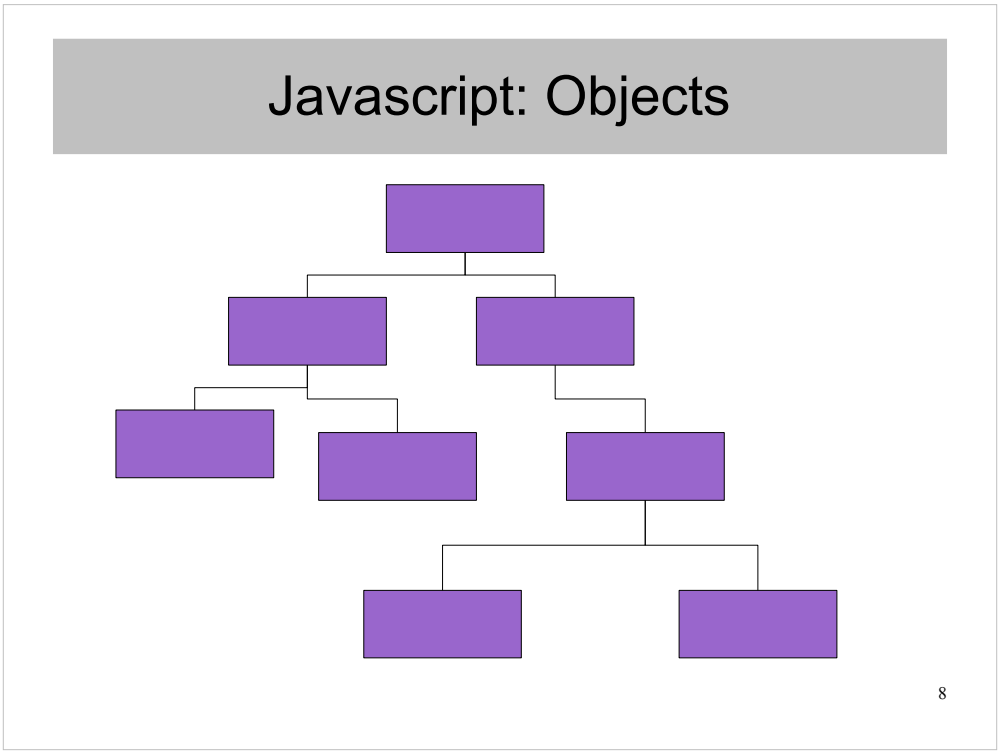
What is at issue here is the inheritance hierarchy, something which appears in some form in most objected-oriented systems.



In many of them, the core objects that form the basis of reuse are called classes. They hold the the common features and specification for a type of object.

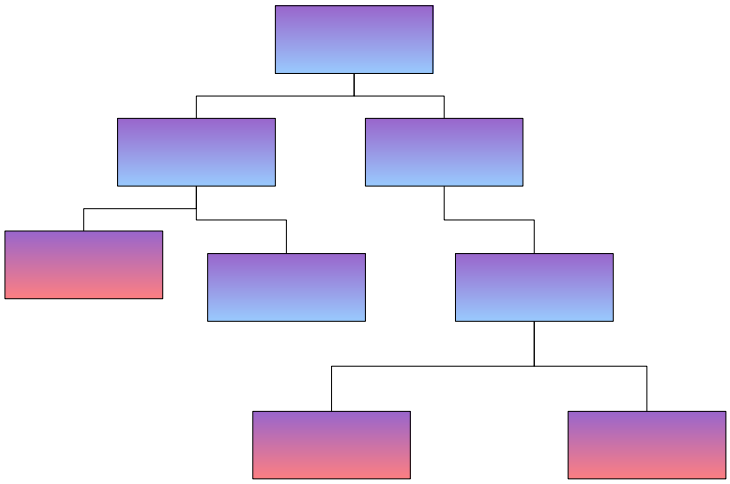


Instances most often hold the specific data, and in many cases are just fancy data structures with all their behavior defined in the class.



But in Javascript, it's “just objects”: the underlying mechanism doesn't really care how it's used.

...Which are often used classically

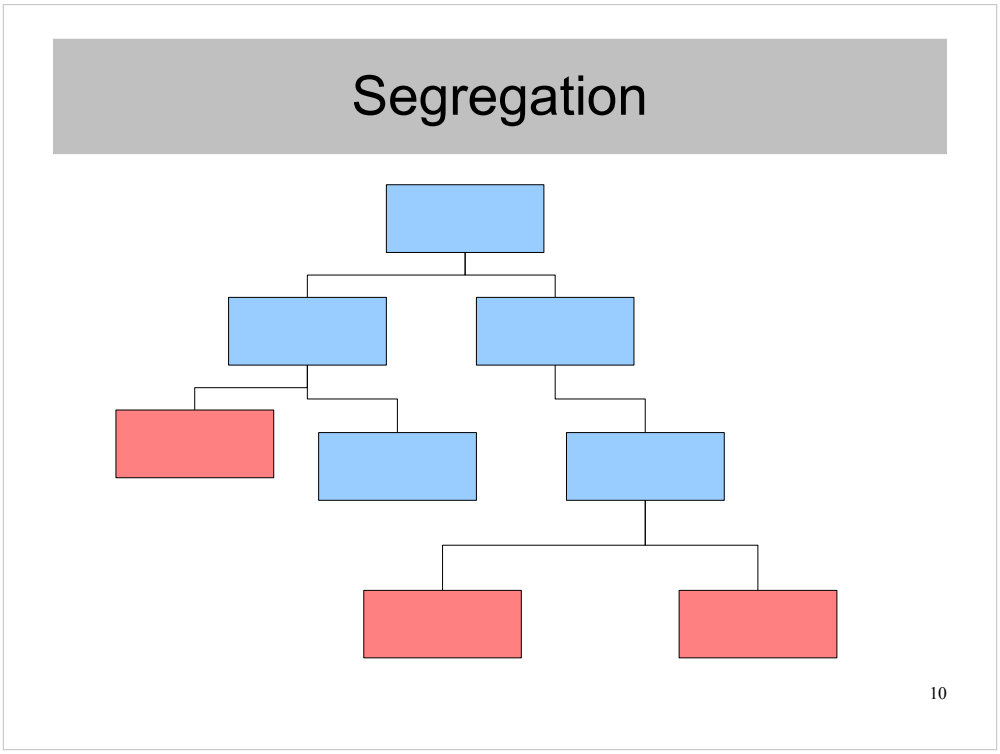


9

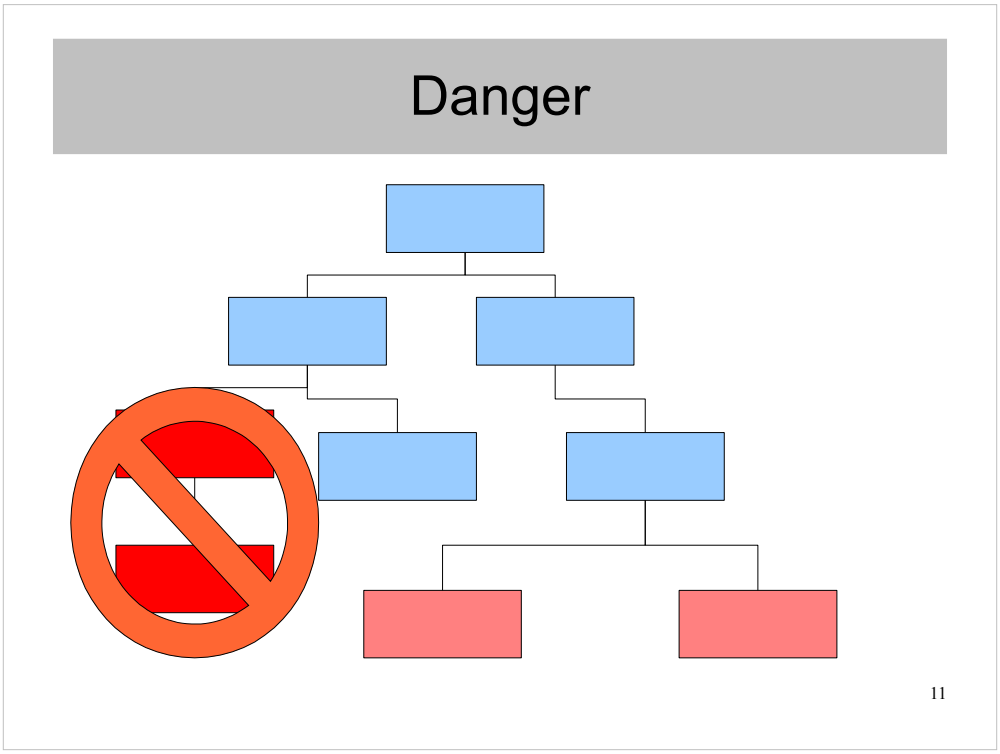
Of course, the class/instance pattern IS useful in many situations, and very often it is how you want to use Javascript.

It just isn't _*always*_ how you want to use it.

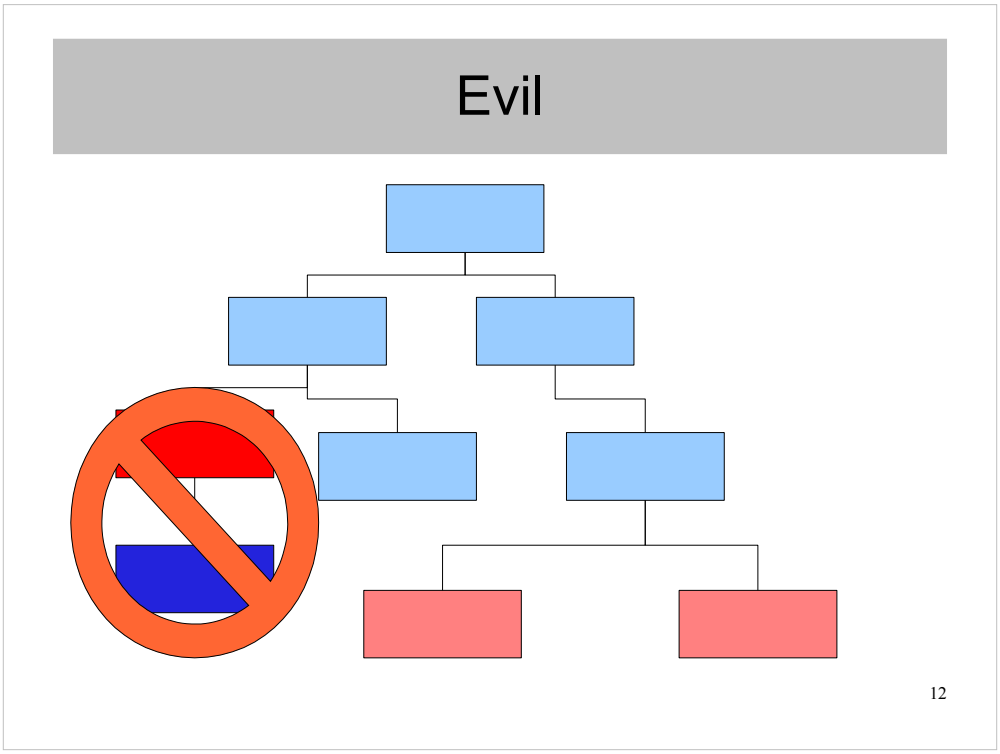
Rigid class systems contribute to segregation...



...Rigid class based systems contribute to segregation.



You couldn't derive an instance from an instance. That would be dangerous...
...So it's not allowed.



And don't even think of deriving a class from an instance. That's just evil...
...and we can't have evil now can we?

The problem is classes are premature optimization...

Click to add title

Classes are Premature Optimization

13

...The problem is classes are premature optimization.

Language authors have forced us to declare static class relationships so that they can more easily optimize programs.

Douglas Crockford's prototypal inheritance.

```
// http://javascript.crockford.com/prototypal.html

function object(o) {
  function F() {};
  F.prototype = (o || {});
  return new F();
}
```

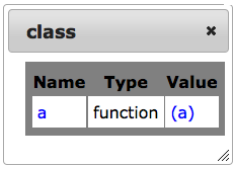
14

So let's talk about how we recover the natural rights of object inheritance.

I, as many people may have, first learned about prototypal inheritance from Douglas Crockford's videos and articles.

He defined a function, initially called 'object', which allows us to make arbitrary derivations easily. I'm going to explain how it works.

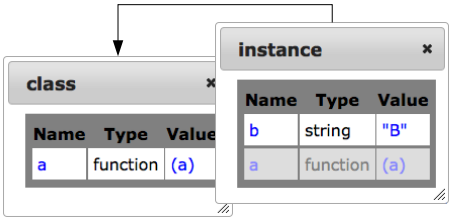
Prototypal



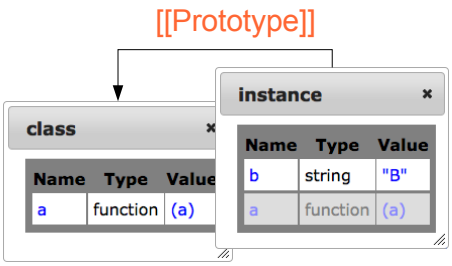
But first I'm going to circle back for a brief review of prototypal inheritance, just to make sure everybody can follow along.

Even though it's prototypal, I'm going to call my objects here 'class' and 'instance' to help you keep in mind their relationship.

Prototypal
inheritance



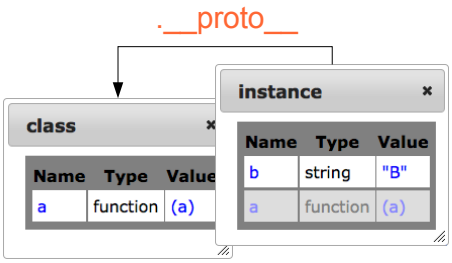
Prototypal
ECMAScript Spec



17

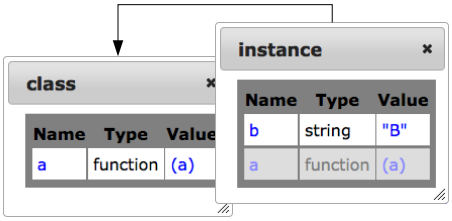
In the ECMAScript (or Javascript) spec, that pointer is called double bracket prototype.

Prototypal
Mozilla

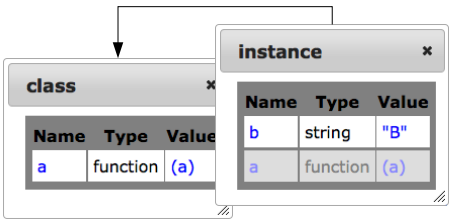


Prototypal ECMAScript 5

Object.getPrototypeOf(instance)

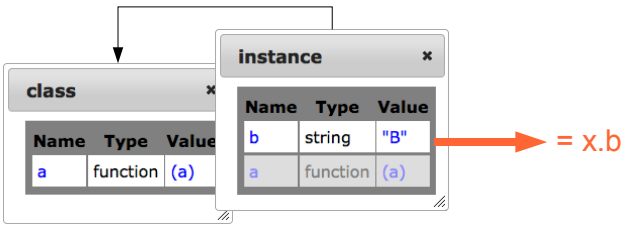


Prototypal

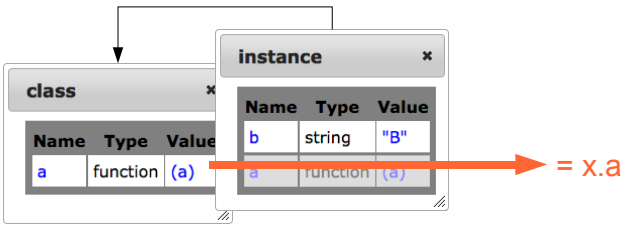


We are just going to ignore the name for now.

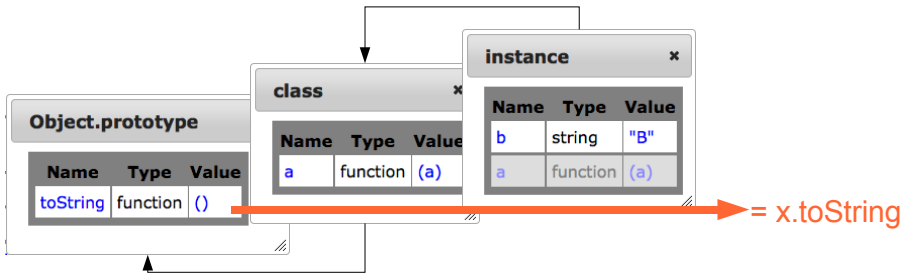
Prototypal
property



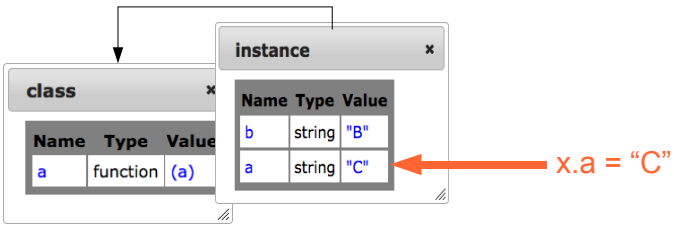
Prototypal
inherited property



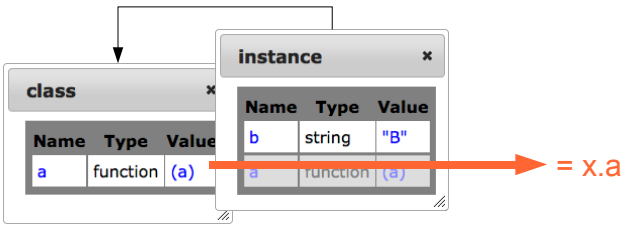
Prototypal
non-enumerable property



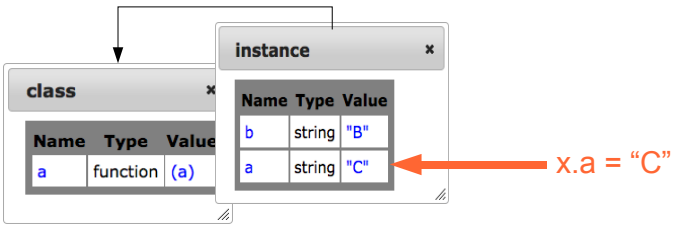
Prototypal
write



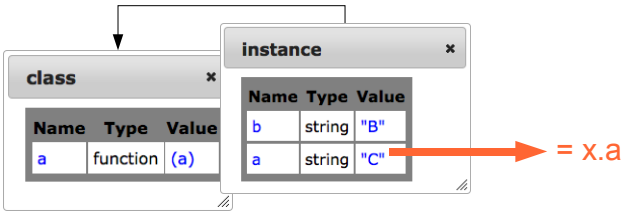
Prototypal inherited property



Prototypal write



Prototypal
masked property



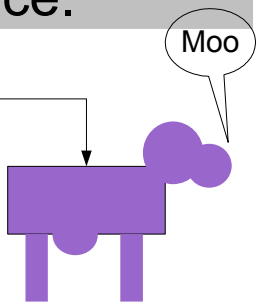
Douglas Crockford's prototypal inheritance.

```
function object(o) {
  function F() {};
  F.prototype = (o || {});
  return new F();
}
```


It takes any object as input. We don't really care what this object is.

It might be a purple cow for all we care....

Douglas Crockford's prototypal inheritance.



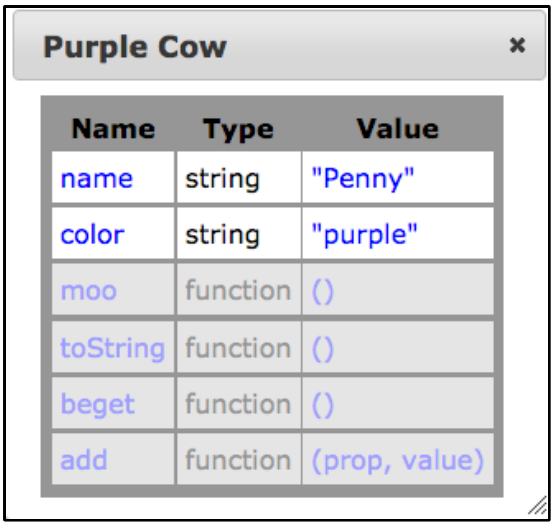

```
function object(o) {
  function F() {};
  F.prototype = (o || {});
  return new F();
}
```

29

...It might be a purple cow for all we care.

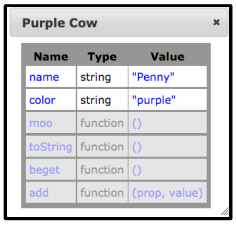
All that matters is that has some particular properties....
...that we want to copy.

Douglas Crockford's prototypal inheritance.



Our purple cow might look like this. It's a pretty classical object – behavior defined by functions derived from it's prototype, and some instance-specific properties.

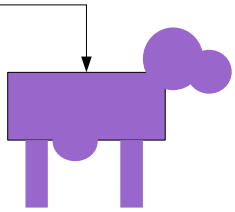
Douglas Crockford's prototypal inheritance.



Just keep that in mind as we progress.

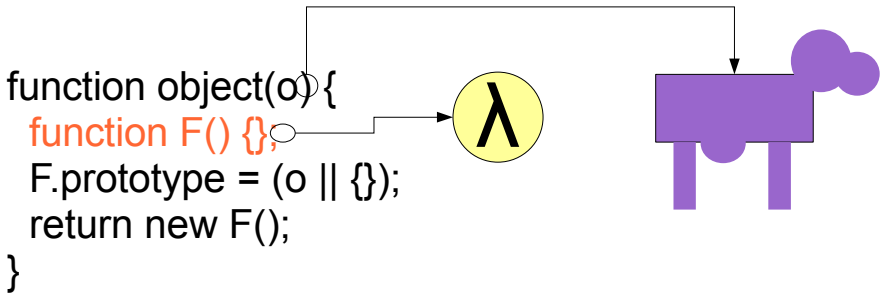
Douglas Crockford's prototypal inheritance.

```
function object(o){
  function F() {};
  F.prototype = (o || {});
  return new F();
}
```



Now, the only way to assign a prototype is to call a constructor function....

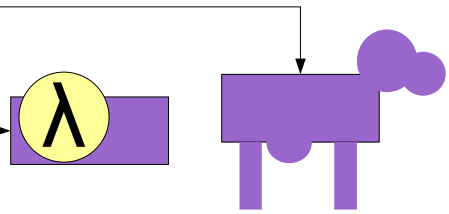
Douglas Crockford's prototypal inheritance.



...So we have to create a dummy function.

Douglas Crockford's prototypal inheritance.

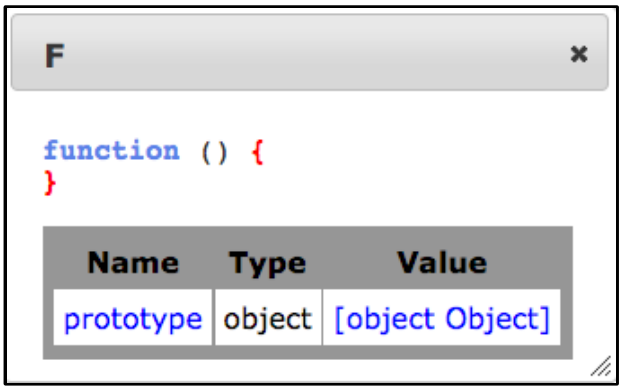
```
function object(o) {
  function F() {};
  F.prototype = (o || {});
  return new F();
}
```



34

Now, as you might be aware, functions, like most things in Javascript are objects.

Douglas Crockford's prototypal inheritance.



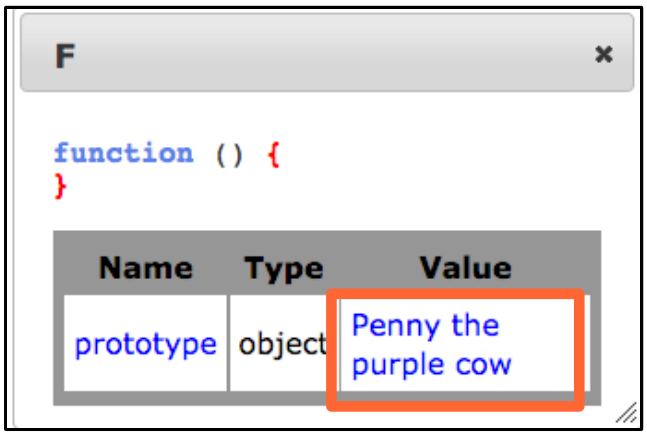
35

You could imagine that our function object looks like this.

Out of the box, it only comes with on interesting property called 'prototype', which initially points to a newly minted, empty object.

But now we tell our function to a have a cow....

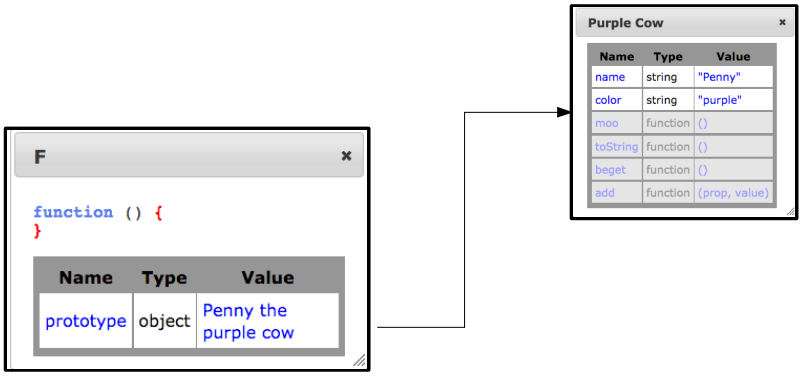
Douglas Crockford's prototypal inheritance.



...Tell our function to a have a cow.

Now remember that whats really going on here is reassigning some pointers...

Douglas Crockford's prototypal inheritance.

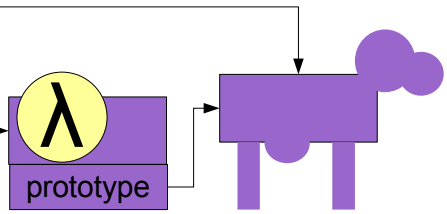


...reassigning some pointers.

So now we've got a constructor for purple cows.

Douglas Crockford's prototypal inheritance.

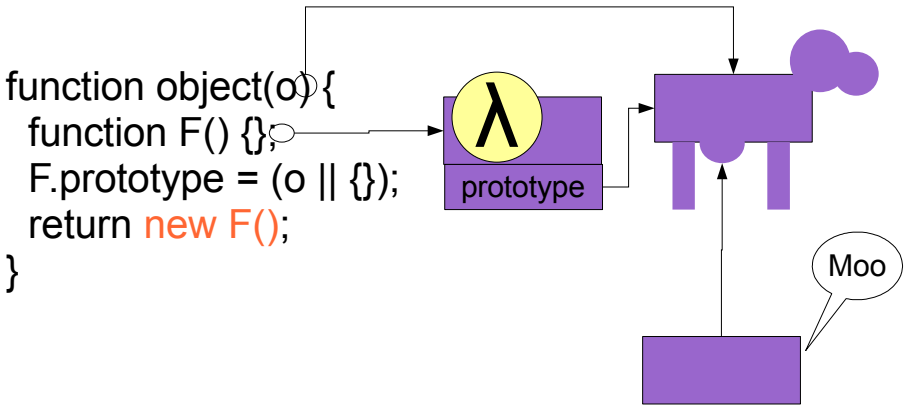
```
function object(o) {
  function F() {};
  F.prototype = (o || {});
  return new F();
}
```



38

The next step is to take our function and milk it...

Douglas Crockford's prototypal inheritance.



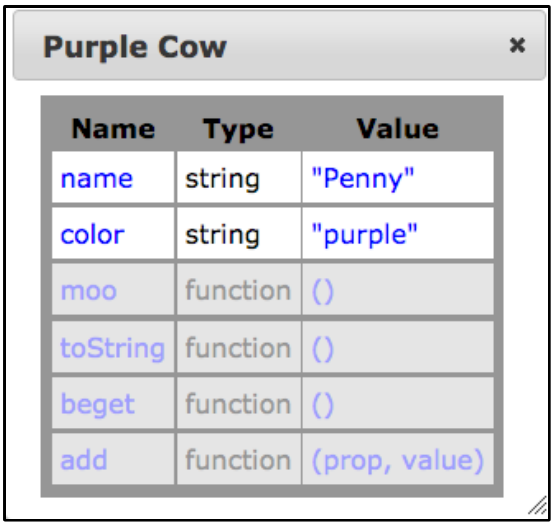
39

...Milk it

Now we've got a new object which looks just like our target object....

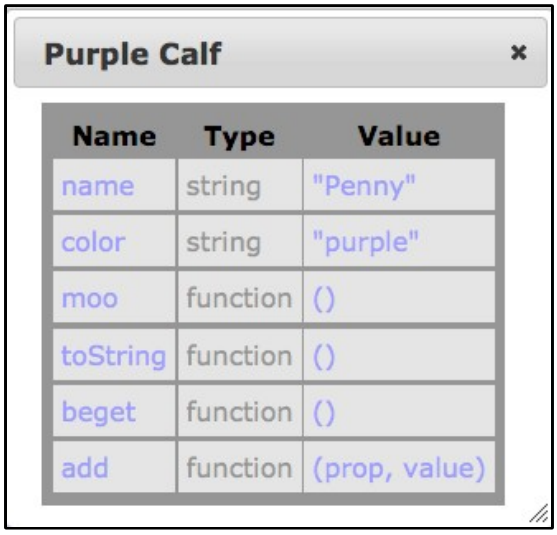
...Unless of course we go asking personal questions with hasOwnProperty.

Douglas Crockford's prototypal inheritance.



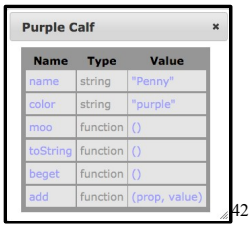
So if our original object looked like this.

Douglas Crockford's prototypal inheritance.



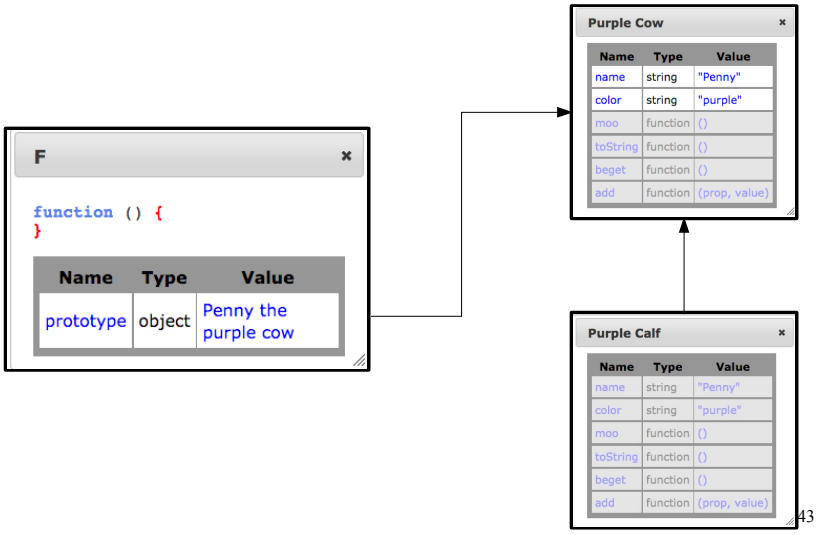
Our new object would look like this. On the surface it's exactly the same as the prototype, except that it doesn't own any of it's properties.

Douglas Crockford's prototypal inheritance.



Putting things in perspective.

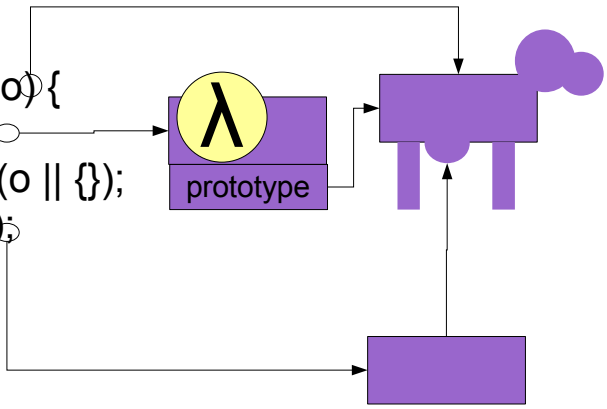
Douglas Crockford's prototypal inheritance.



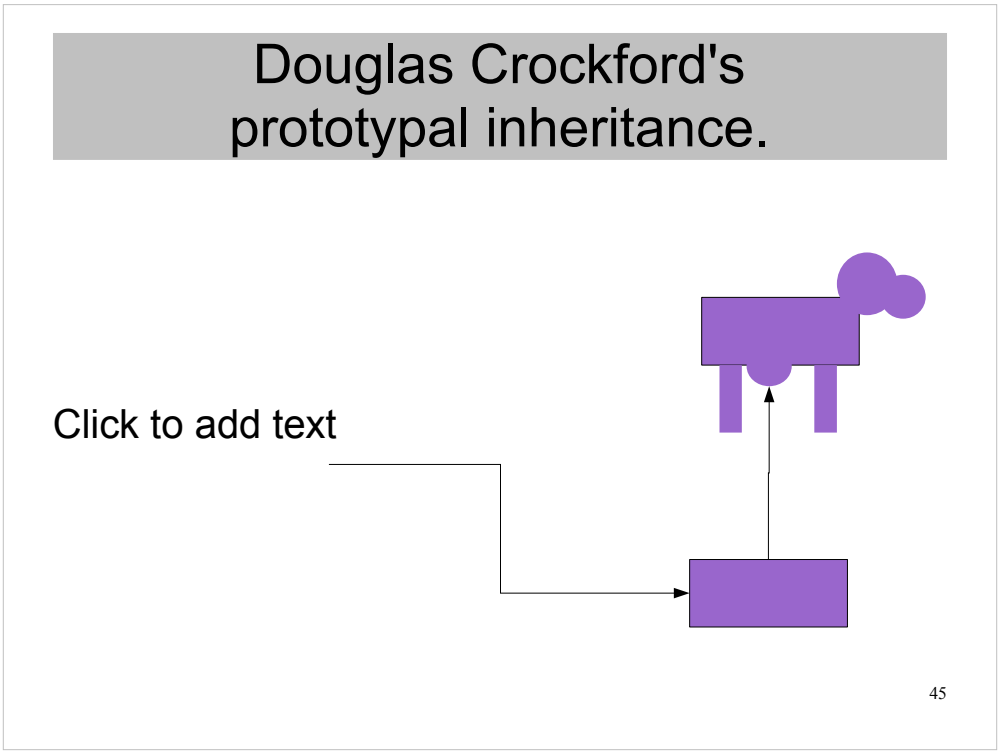
The prototype of course is just a pointer, or at least we can think of it that way.

Douglas Crockford's prototypal inheritance.

```
function object(o) {
  function F() {};
  F.prototype = (o || {});
  return new F();
}
```



Once we've constructed our object, all we have to do is return it...



...Leaving a reference to the new object, which has an indirect reference to our prototype object.

The function and company are grist for the garbage collector.

Disclaimers

Not the only way.

Disclaimers

Not necessarily the best way.

Disclaimers

Know how it works,

So you can recognize when it might be useful.

Click to add title

Photo: hmboo http://www.flickr.com/people/kefraya/

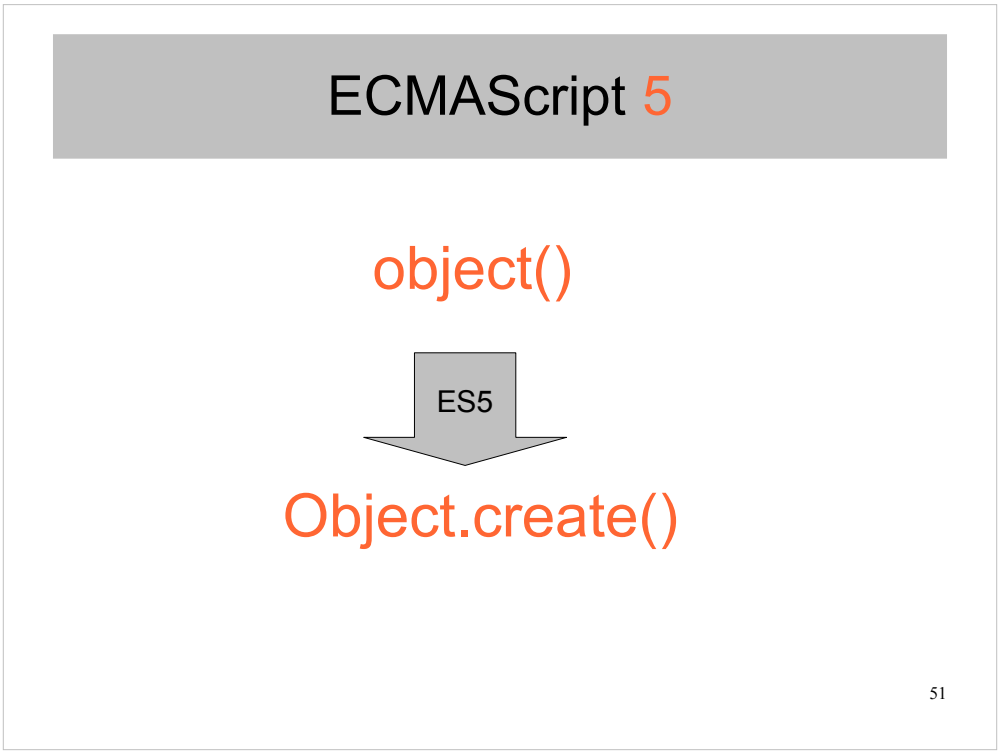
In short, I'm no trying to tell you that prototypal inheritance is a hammer for all you nails.

Click to add title

50

Photo: hmboo http://www.flickr.com/people/kefraya/

What I am trying to convince you of is that classical Inheritance is often a hammer, and I'm asking you to broaden your tool belt a little bit.



One thing I should note is that ECMAScript 5 recently got standardized. It includes an enhanced version of the 'object' function available as Object.create. I'll be using that syntax for all further examples.

Example: tick marks

52

First example.

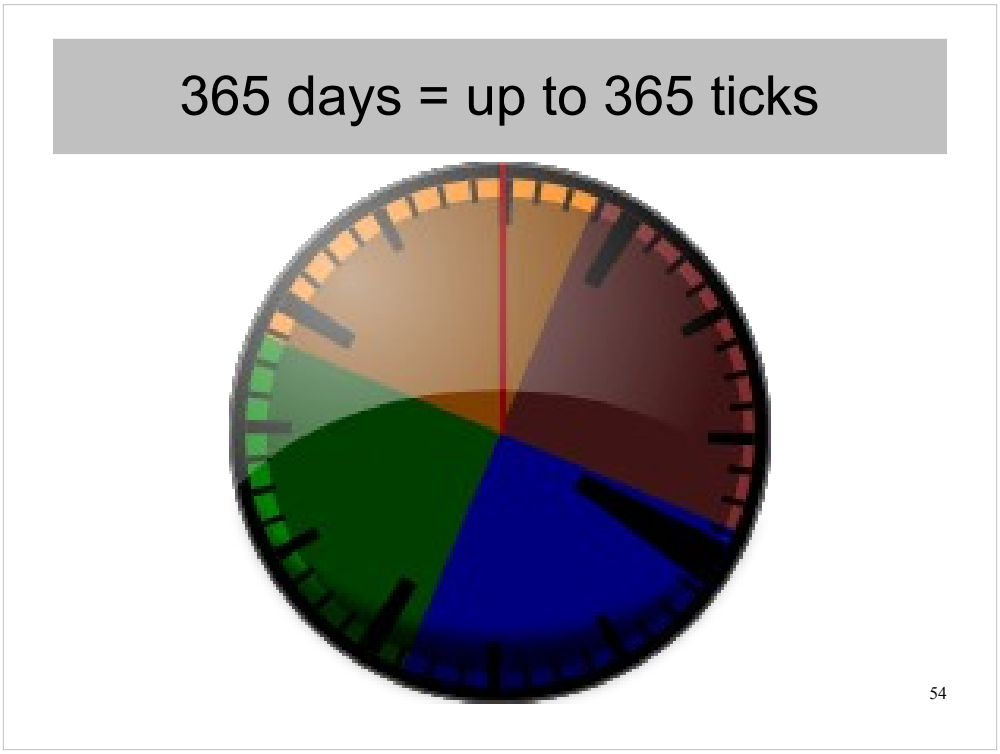
I actually got into Javascript though OS X Dashboard widgets. I figured I'd be learning Objective-C and other Apple-specific APIs, but discovered that dashboard widgets are really little web pages.

Calendar

53

Now my clock can have a many different sets of disks.
One of those sets is a calendar of sorts.

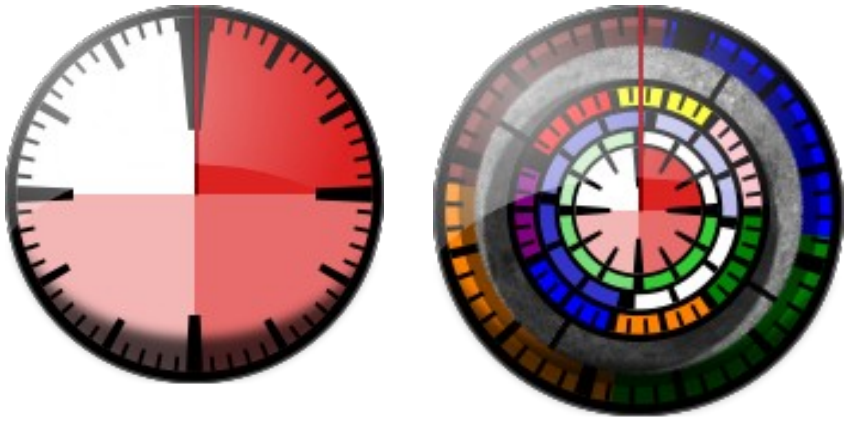
Of particular interest is the year disk....



...the year disk.

The problem is that there can be up to 365 tick marks, but as you can see, rather less are actually drawn, a fact I discovered by a spammed debug console as the drawing routine considered each possible location.

Scaling



55

The main feature complicating matters is scaling. Each disk can be drawn at nearly infinite different sizes.

It may not be obvious, but the two red disks here are the same.

Marks change with size

56

It's easier to see if we zoom in.

However, now you can hopefully see that the some tick marks have changed size as the disk shrinks, while others have disappeared to avoid getting too crowded.

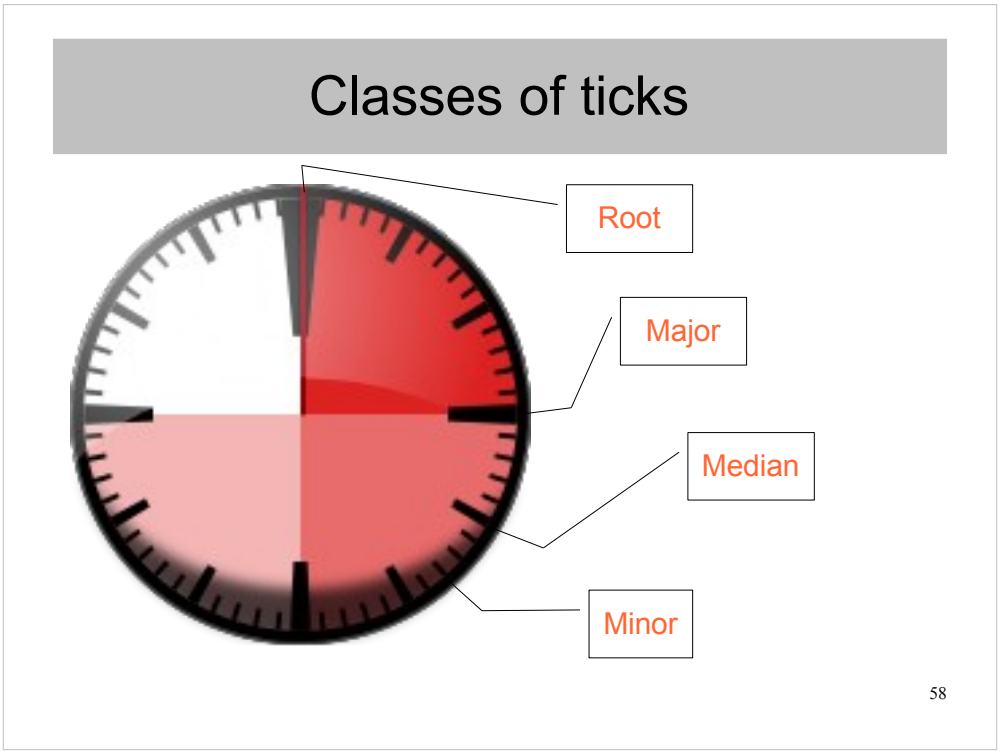
Now, you are all probably thinking. “Justin, this all very interesting, “....

What does this have to do with prototypal inheritance?

57

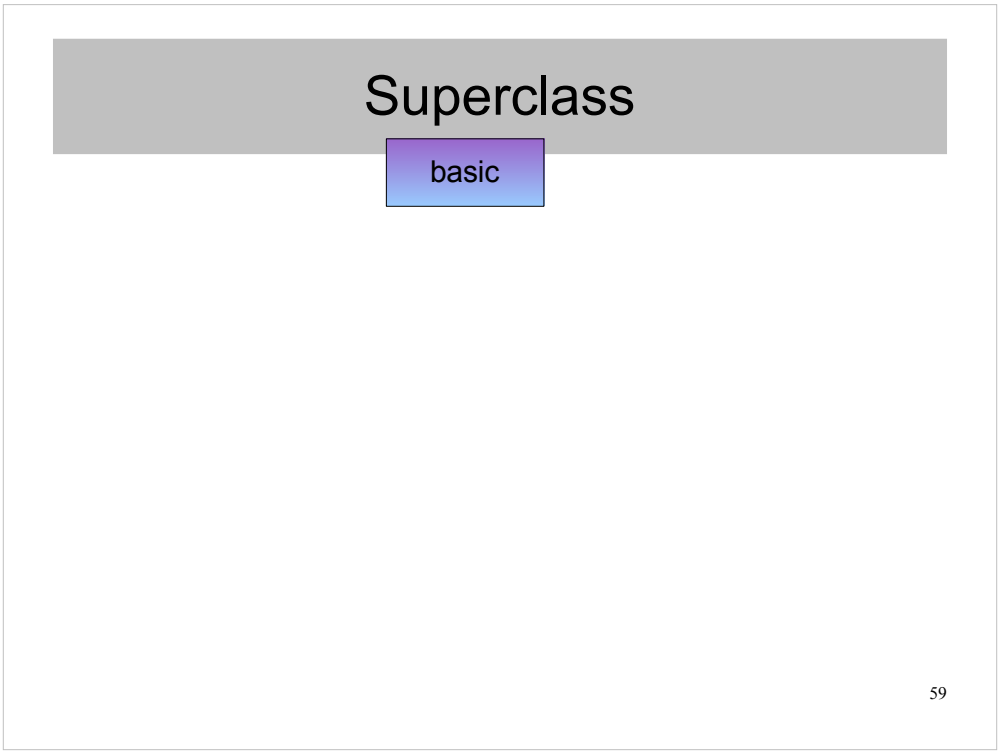
....“but what does this have to do with prototypal inheritance”

Well, here's the thing...

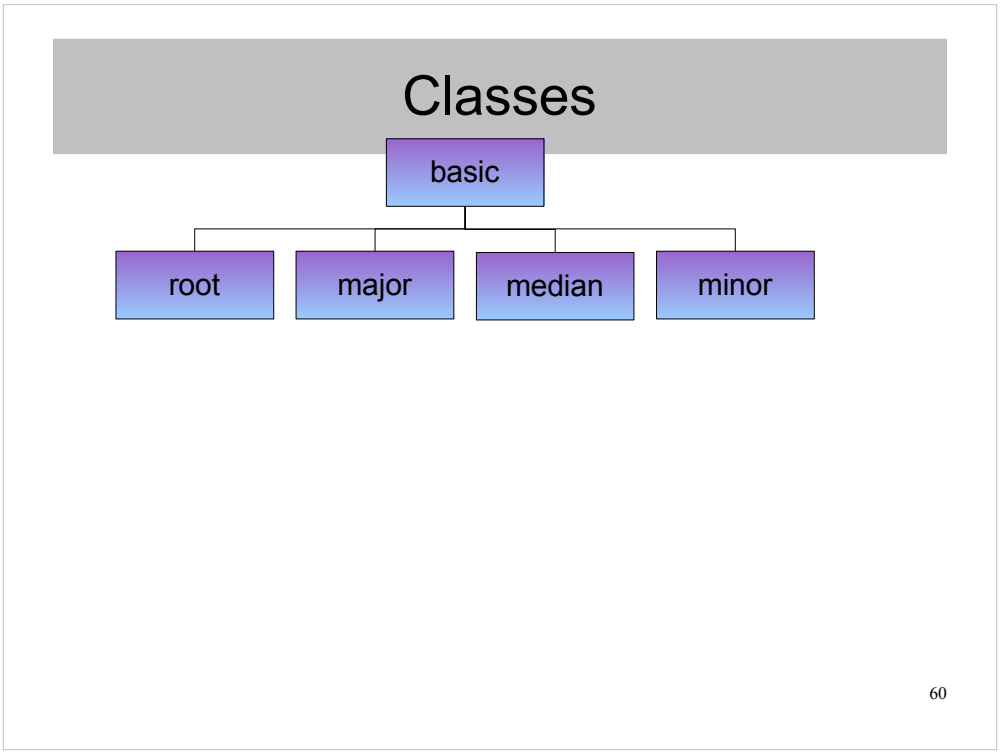


All the ticks on a disk fall into a fairly limited set of groups... you could even call them classes.

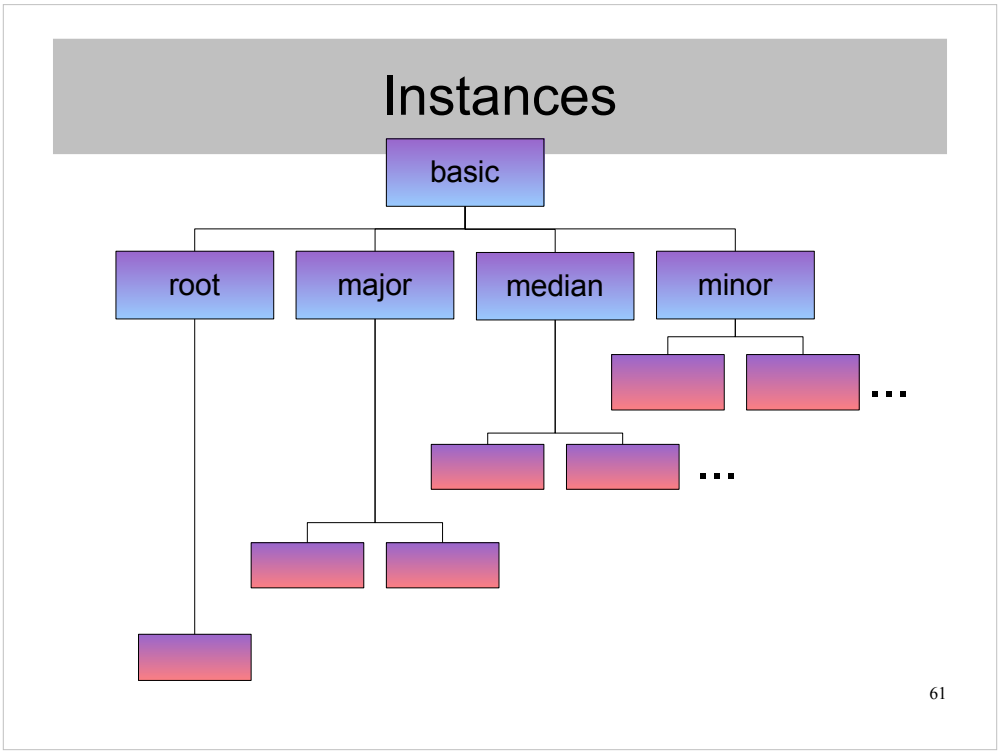
I called my classes root, major, median and minor.



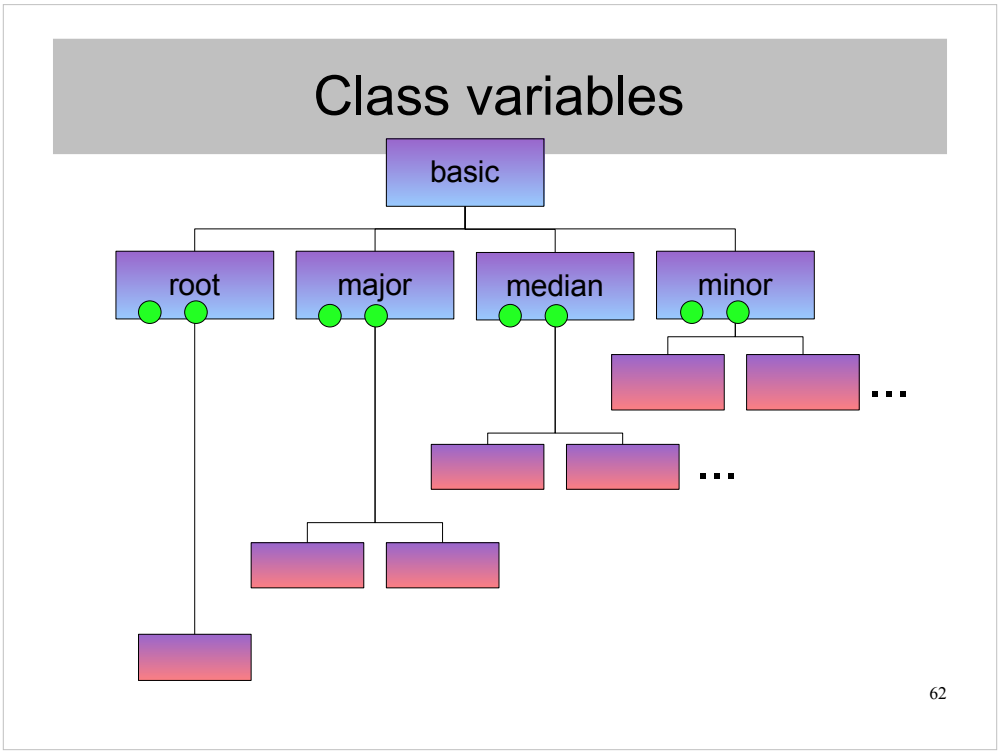
All ticks on a given disk have a common ancestor to tie them together.



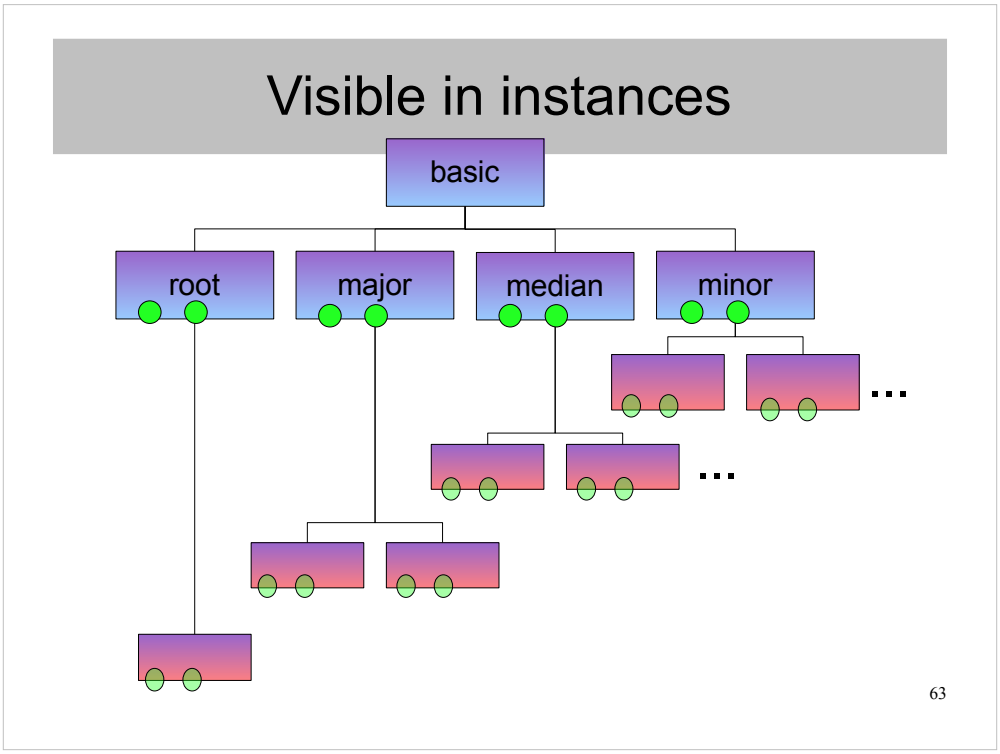
From there I derive the classes I need.



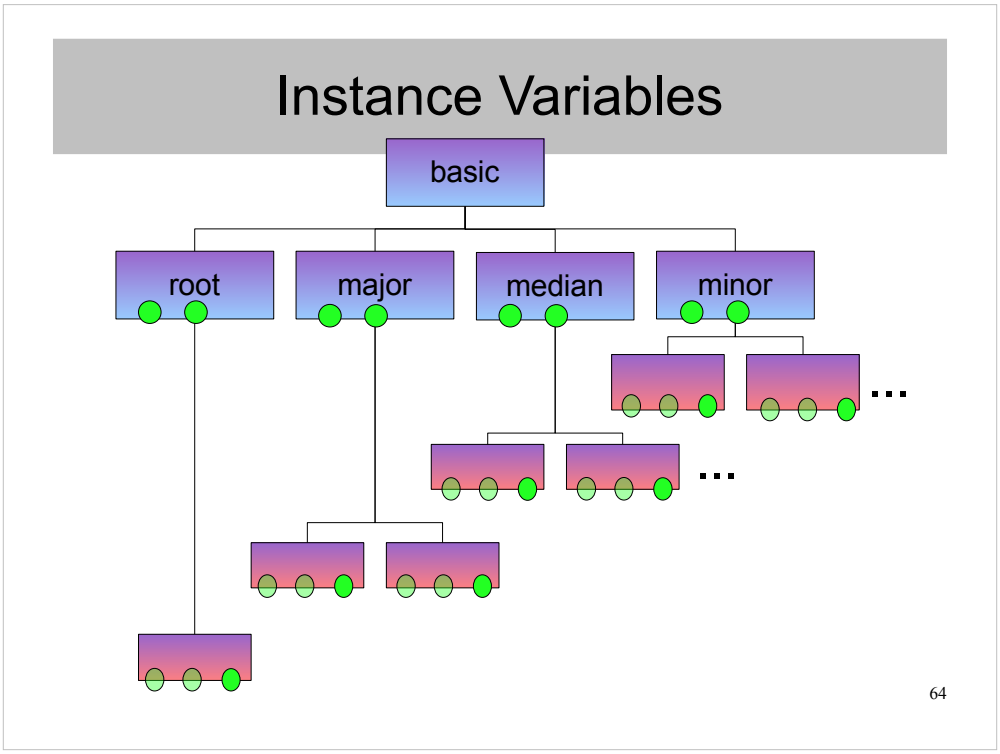
From which are derived the individual tick marks themselves.



The tick classes take care the common properties, basically features of size.

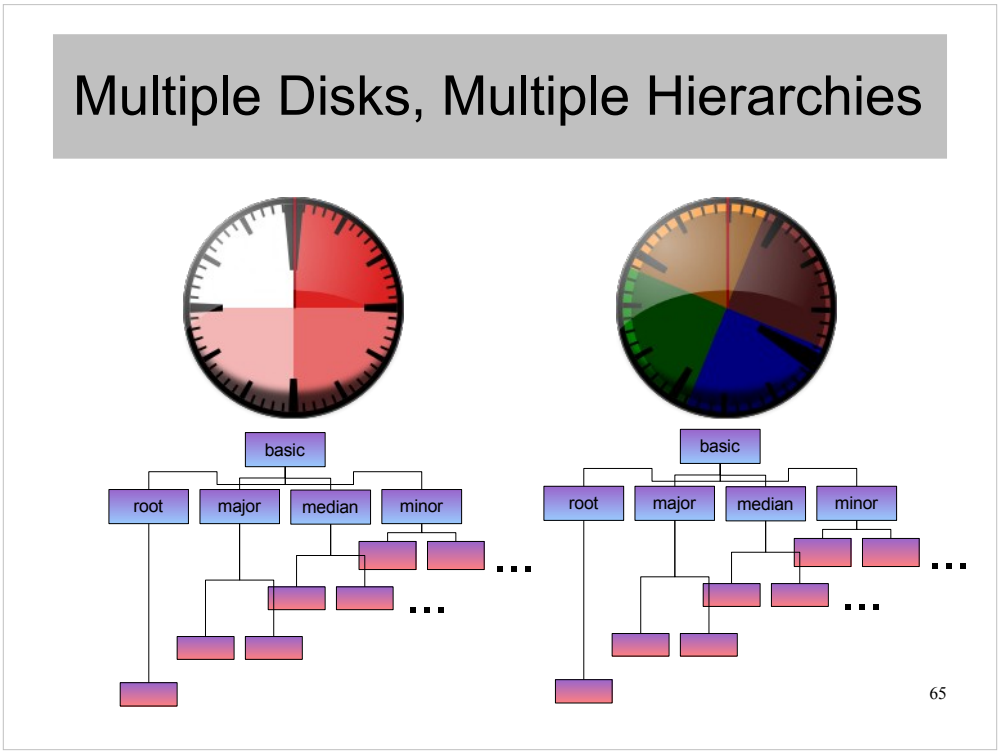


These properties of course are visible on each instance....



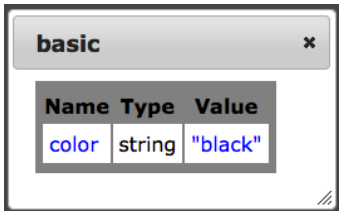
... which adds it's own more specific data.

Now, you might be thinking that this looks like a pretty ordinary use of classes and instances. Don't forget however, that I have multiple disks...

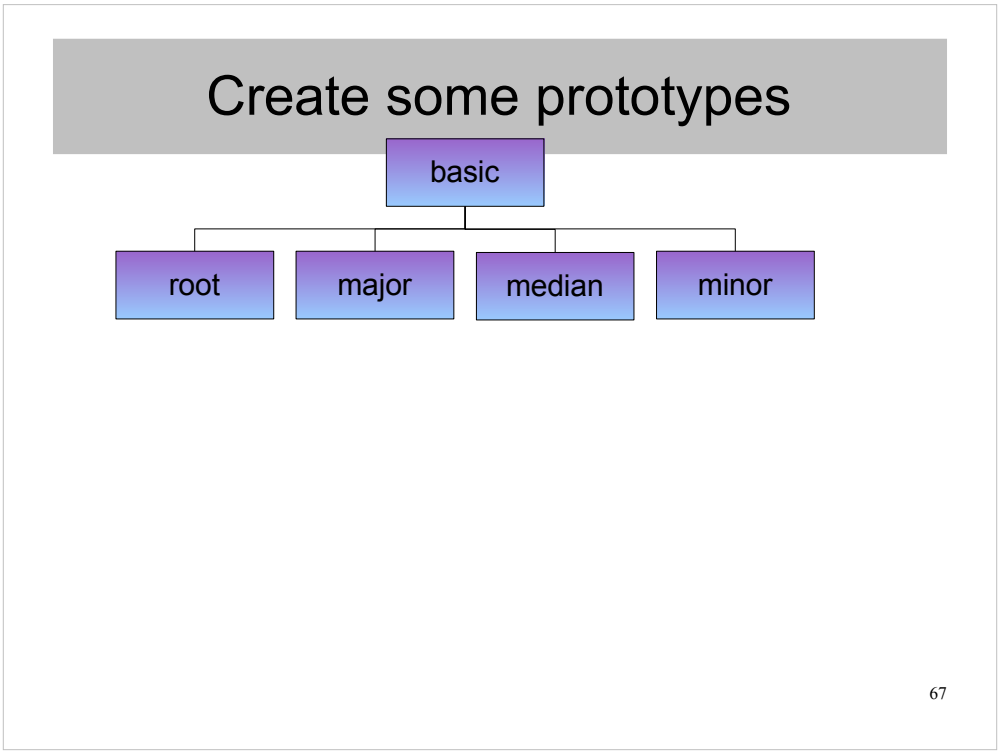


...Each of which has it's own classes and instances. If it came down to generating these by hand I wouldn't have used anything like this. But with a dynamic, prototypal language, it was a fun exercise.

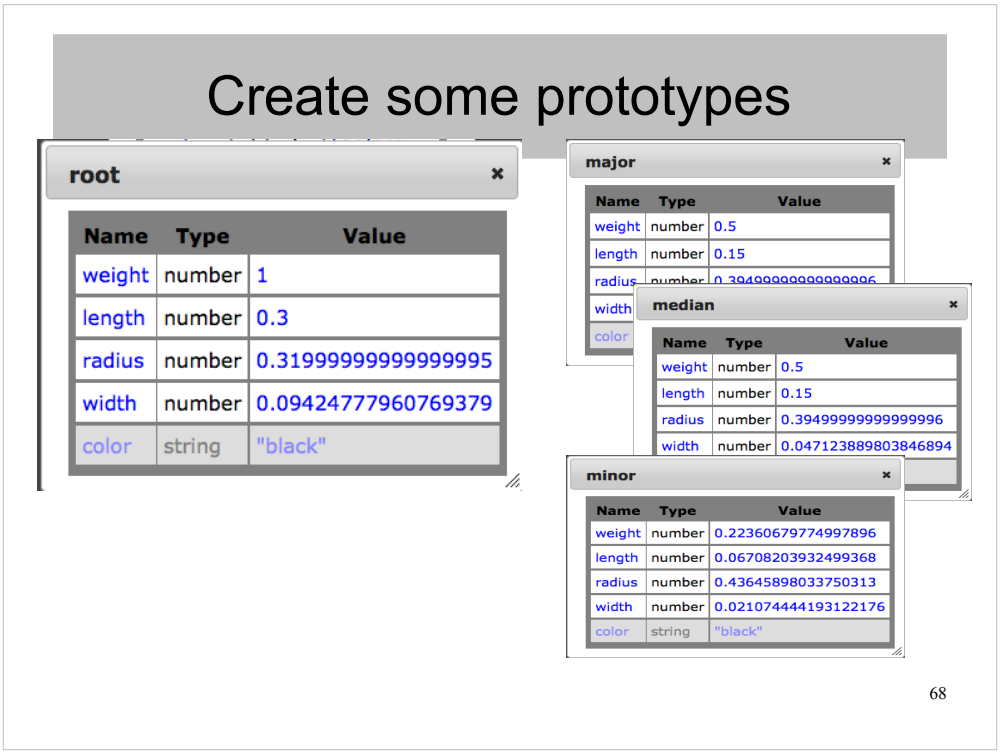
Superclass



The only thing my super class says is that the default color is black.



Now I specialize a little more.

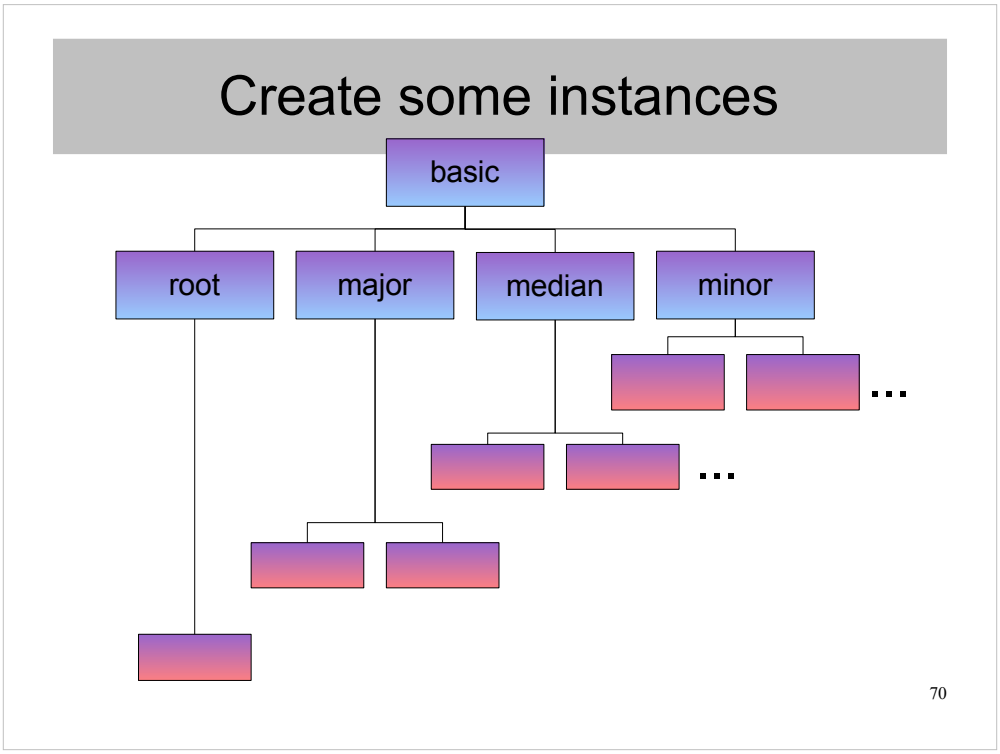


The class objects inherit the base property and add properties for length, width, and position from center.

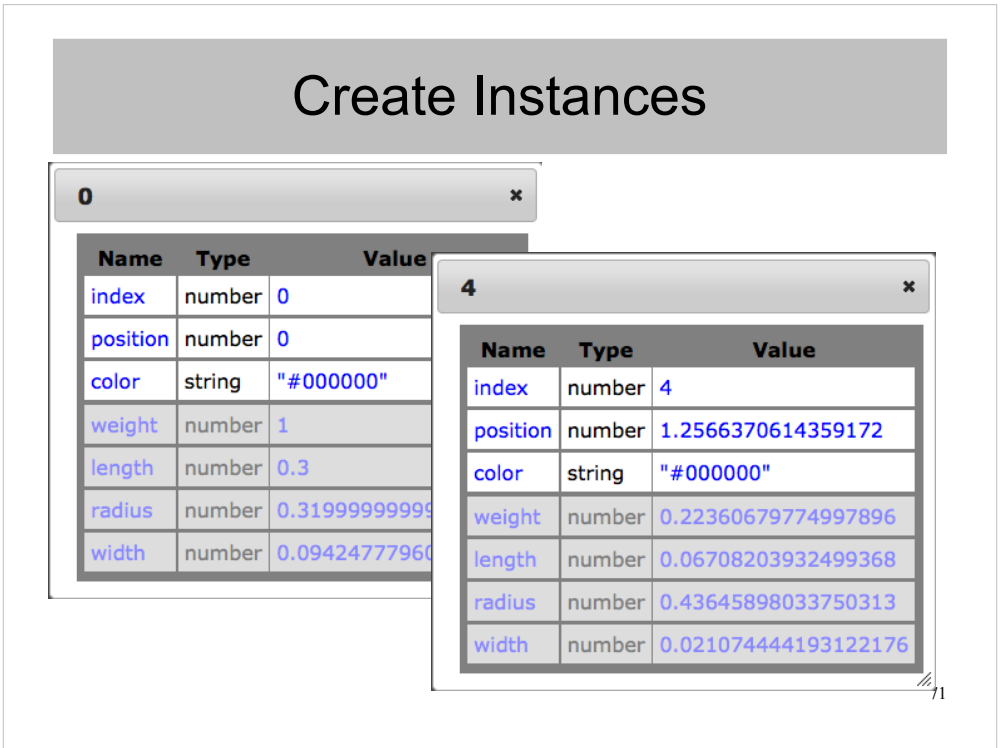
Create some prototypes

```
var basic = {color: 'black'};
this.markers = {
  basic: basic,
  root: Object.create(basic),
  major: Object.create(basic),
  median: Object.create(basic),
  minor: Object.create(basic),
  marks: []
};
```


Of course, I use the prototypal constructor to create them from the basic tick.



The next step is too create some instances.



Instances share all the size data from the classes and add more specialized information for the exact position.

One thing I discovered while putting this together is that I'm always overriding the basic color, evan if it's still black. A minor problem I'll have to revisit later.

Create instances

```
for (var i = 0;i < this.markerCount;i++) {
  var mark = this.createMarker(i);
  if (mark) {
    this.markers.marks.push(mark);
  }
}
```


Now, if you recall the point of all this was to avoid dealing with invisible marks. This is handled by having a mark factory, which returns null if the slot should be empty.

Create instances

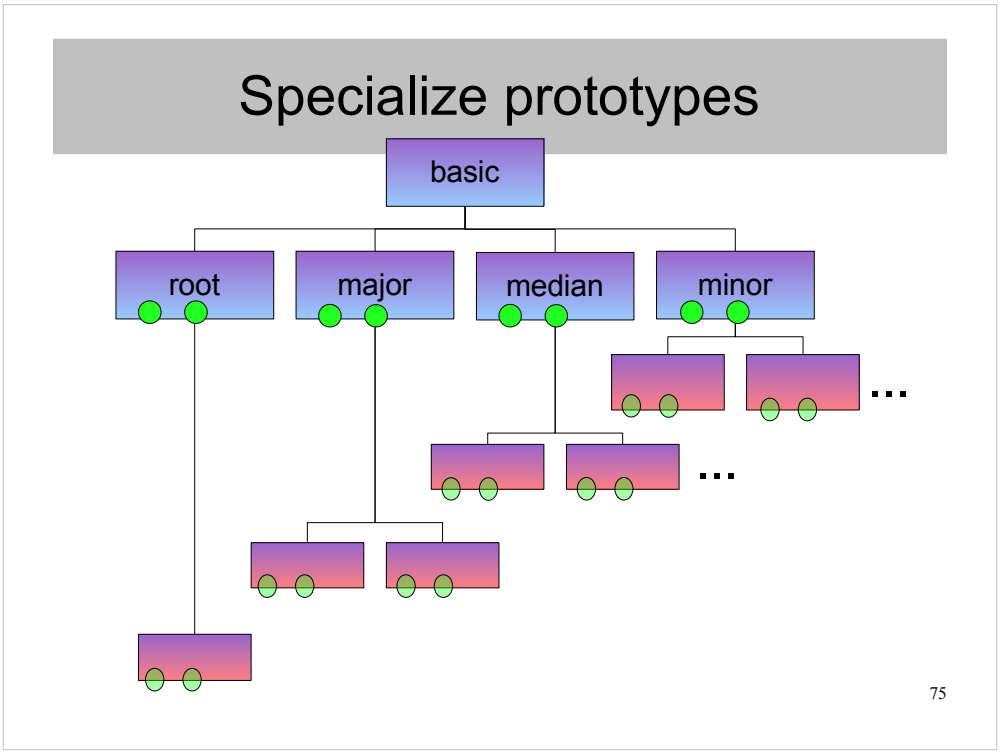
```
return mix(Object.create(kind), {
  index: i,
  position: RADIANS * i / this.markerCount,
  color: this.markerColor(i)
});
```


Inside that factory, I use the object constructor again to derive an instance from the appropriate class.

Create instances

```
return mix(Object.create(kind), {
  index: i,
  position: RADIANS * i / this.markerCount,
  color: this.markerColor(i)
});
```


After that, I use one of my own utility functions to mix in a bag of it's specific properties to specialize.

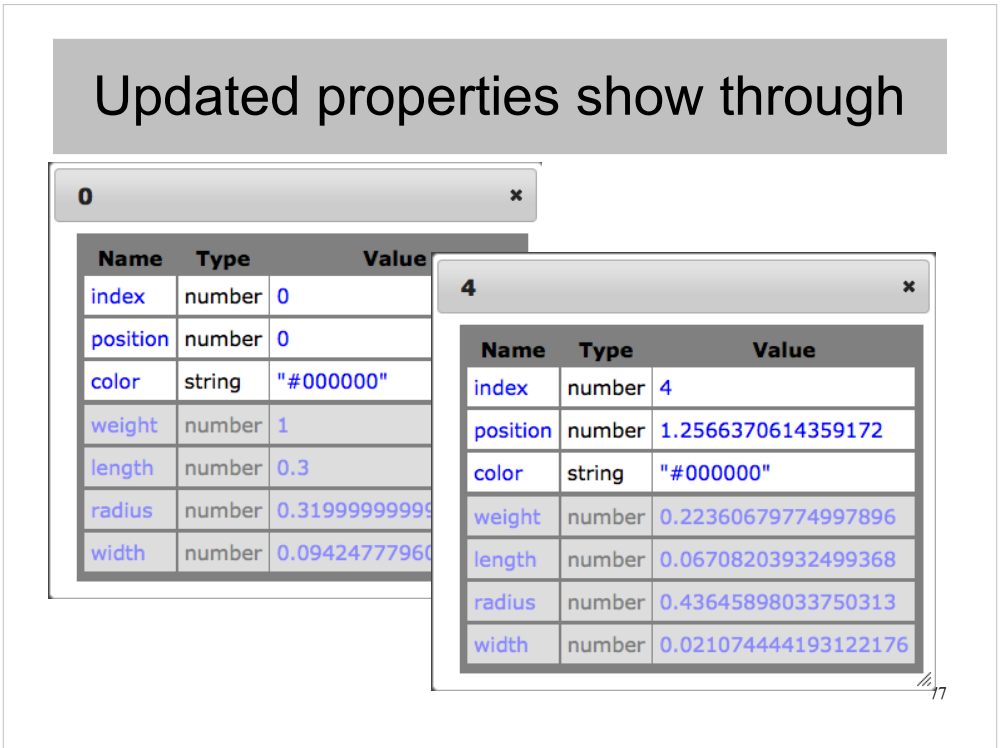


Now, the purpose of this exercise was to take advantage of the common properties in the tick classes.

Specialize prototypes

```
mix(this.markers.root,
  this.markerSize(this.rootWeight));
mix(this.markers.major,
  this.markerSize(... * this.majorWeight));
mix(this.markers.median,
  this.markerSize(... * this.medianWeight));
mix(this.markers.minor,
  this.markerSize(... * this.minorWeight));
```


When the classes are created, I mix in a bag of properties (from a helper function) that calculates the appropriate size.



Any time I update those properties, the new values show through in all of my instances.

Resizing doesn't loop

```
resizeMarkers: function() {
  mix(this.markers.major, ... );
  mix(this.markers.median, ... );
  mix(this.markers.minor, ... );
},
```

78

Every time a disks resizes (which might be quite often, since resizing is animated) I only have to update the classes, not each instance.

Drawing is easy

```
context.lineWidth = mark.length;
context.strokeStyle = mark.color;
context.beginPath();
context.arc(0, 0, mark.radius,
  mark.position - mark.width,
  mark.position + mark.width,
  ARC.CLOCKWISE);
context.stroke();
```


When it comes time to draw, I don't have to iterate over every possible mark location, and I don't have to care whether a property is coming from a class or an instance – it's just object member access.

Tick marks: Patterns

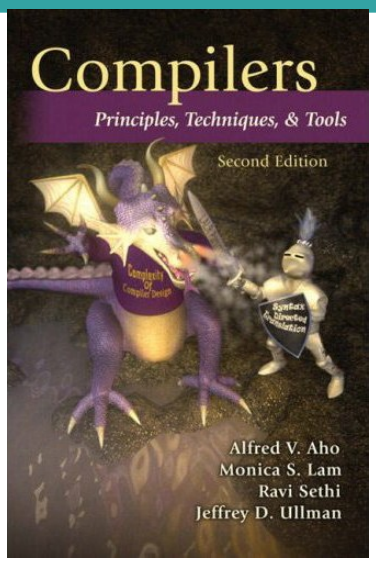
Shared data

Flyweights

The main pattern at play here is shared data, with the common size properties in the tick classes.

There is a small aspect of flyweight, although they are somewhat fat flies with several properties on each instance.

Example: parser state



81

Next up is a dungeon crawl in state machine.

For a long time, Disk Clock's various faces were defined by long multiplications of constants to calculate the number of ticks or how much time each one represented.

Among the things that fascinate me are domain specific languages, and I thought this would be a good place for a limited engagement that could greatly clarify what was going on.

Time expression langauge

- days/year -> 365.24219
- ms/day -> 86400000
- years/(2^31 seconds) -> 68.05110552758107

So I came up with a time expression language that allowed me to write values as I thought of them and let the computer worry about the math.

Of course, I had to worry about parsing.

Time expression language

```
exp = exp ' '+ exp
      | exp/exp
      | exp^exp
      | (exp)
      | ID
      | N
```


I should point out that this is very much an expression language, and far from a general purpose tool.

The nice thing is that this makes the task very simple – I only have one kind of expression and a handfull of ways to form it.

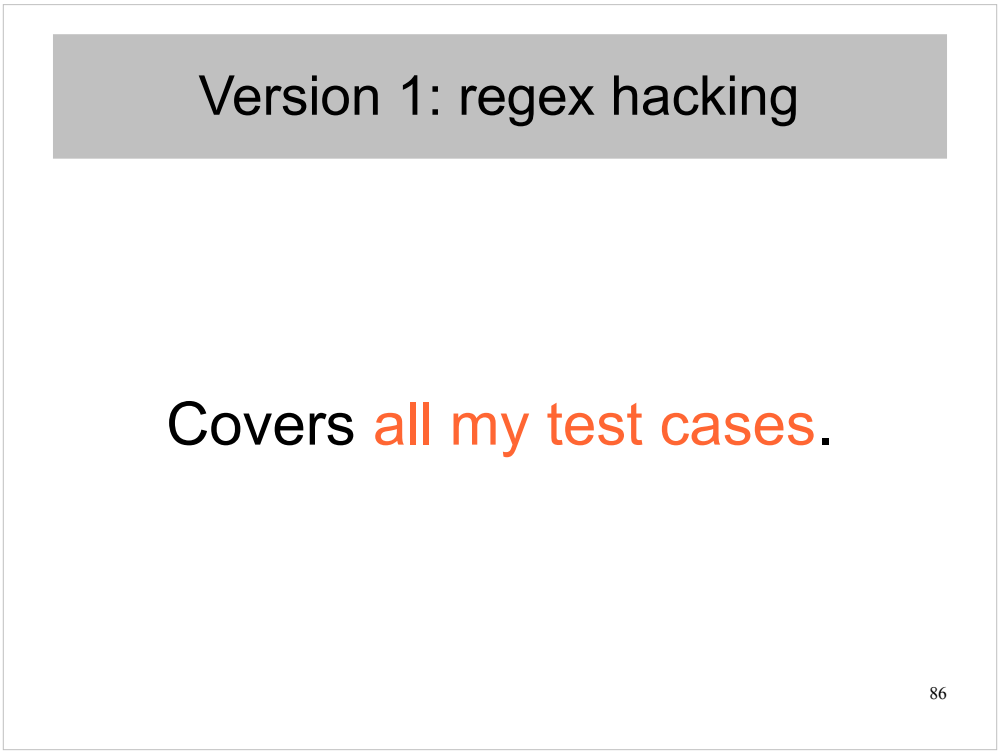
Version 1: regex hacking

About one evening.

So simple, in fact, that I could do it with regex hacking.

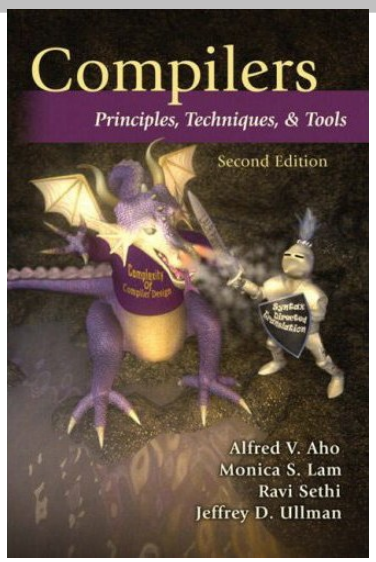
Version 1: regex hacking

40 lines of code.



Covers all the test cases I built up by copying my long multiplications and comparing them to the expression I wanted to use instead.

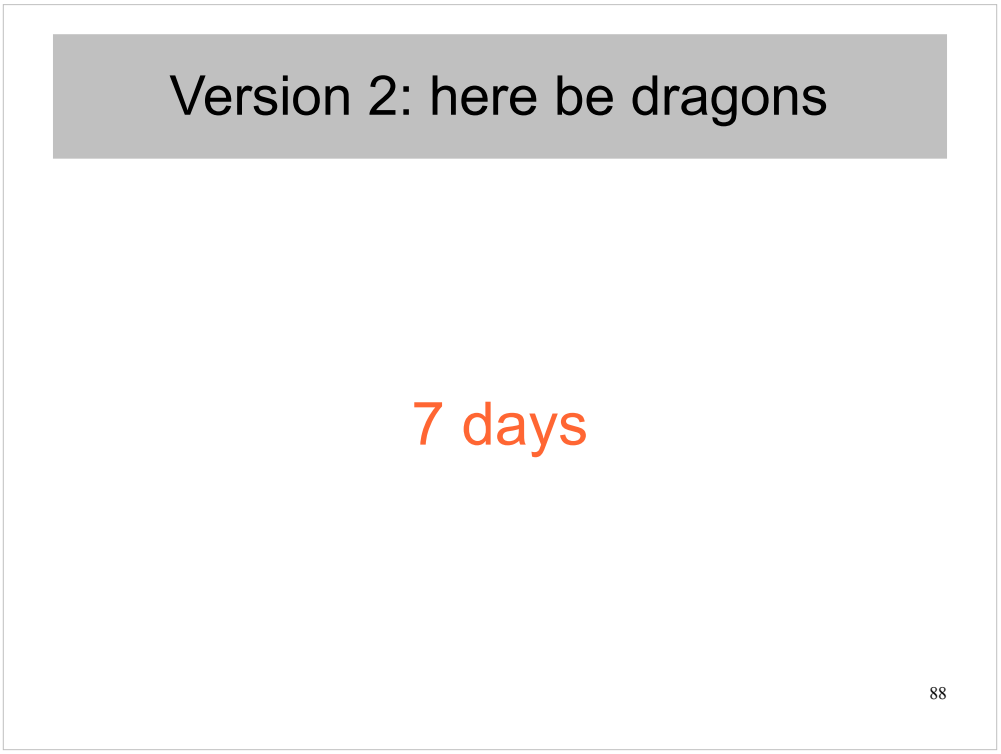
Version 2: More Power!



87

But of course, I had a new book to play with.

The problem with the dragon book is that, here be dragons...



...here be dragons.

In total I was working on the new version on about 7 different days.

Version 2: here be dragons

~900 lines of code
(with spaces, comments, etc.)

Version 2: here be dragons

Not yet used for anything bigger.

A learning experience.

Of course, one of the things I learned was don't do it unless you have to.

Now, you may be asking “Justin, this is all very interesting, ...”

What does this have to do with prototypal inheritance?

...“but what does it have to do with prototypal inheritance?”

Productions

exp -> exp '/' exp

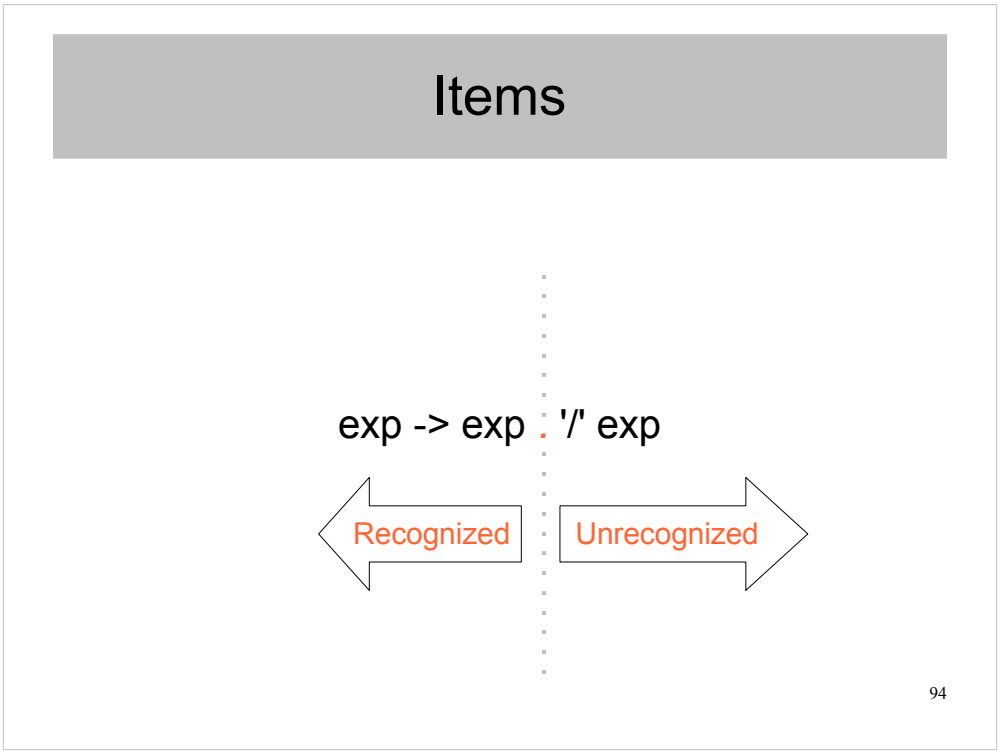
The book's algoritm is based around productions, or ways to form one token from a set of others.

Items

exp -> exp . '/' exp

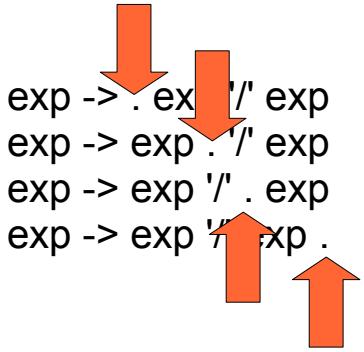
Dot

From productions I define items,which is just a production with a dot.



The dot represents the boundary between the recognized and unrecognized tokens in the expression.

Items



In order to build a state machine, I have to create the closure, or all possible items in every production.

Production

production ×

Name	Type	Value
head	string	"(start)"
pattern	object	exp,(end)
action	function	(v)
precedence	number	0
assoc	undefined	undefined
dup	function	()
mint	function	(head, pattern, action)
assign_precedence	function	(classifier)
toString	function	()
equals	function	(b)
item	function	(d)
complete	function	()
next	function	()

My productions are pretty ordinary classical objects, with a class providing a bunch of methods and data defined on each instance – although one of these pieces of data is a function.

Item

item ×

Name	Type	Value
dot	number	0
head	string	"(start)"
pattern	object	exp,(end)
action	function	(v)
precedence	number	0
assoc	undefined	undefined
dup	function	()
mint	function	(head, pattern, action)
assign_precedence	function	(classifier)
toString	function	()
equals	function	(b)
item	function	(d)
complete	function	()
next	function	()

My item is, much like the real item, a production with a dot.

production.item

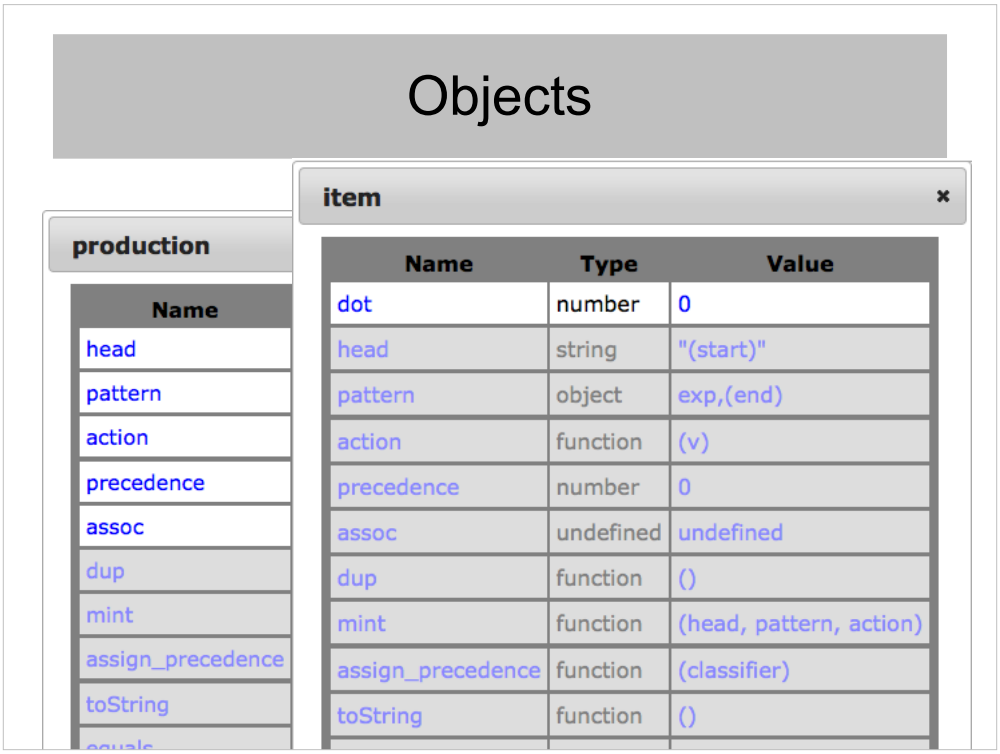
```
item: function(d) {
  var i = Object.create(this);
  i.dot = d;
  return i;
},
```


One of the methods of a production is the method 'item', which calls the object constructor with the current object as prototype...

production.item

```
item: function(d) {
  var i = Object.create(this);
  i.dot = d;
  return i;
},
```


...and then adds the dot property.



The result is an item with full and easy access to it's production, but a minimal amount of extra data to be stored.

Create on demand

- add(production.item(0));
- add(production.item(production.dot + 1));

This method makes it really easy to add items on demand.

Create on demand

- add(production.item(0));
- add(production.item(production.dot + 1));

Since an item 'is-a' production, it's also easy to advance the dot by calling the item method. This ensures that I have an independent object and don't get burned by mutable state on the dot field.

Parser state: Patterns

Shared data

Flyweights

Near-miss

This example once again used shared data – items wanted access to all of the production data.

It's a much better example of flyweight, with minimal new data required on each item.

I also think of it as a near-miss. Items needed so much of a production, why not just re-use it directly?

Patterns

Shared behavior

Shared data

Flyweights

Near-miss

Sparse objects

Nested Context

Now I'm going to talk about some of the situations where you might want to use prototypal inheritance. Some of these come from the previous examples, others don't.

Shared behavior

i.e, functions/methods

Shared behavior

Classical inheritance

Shared Data

Tick Marks

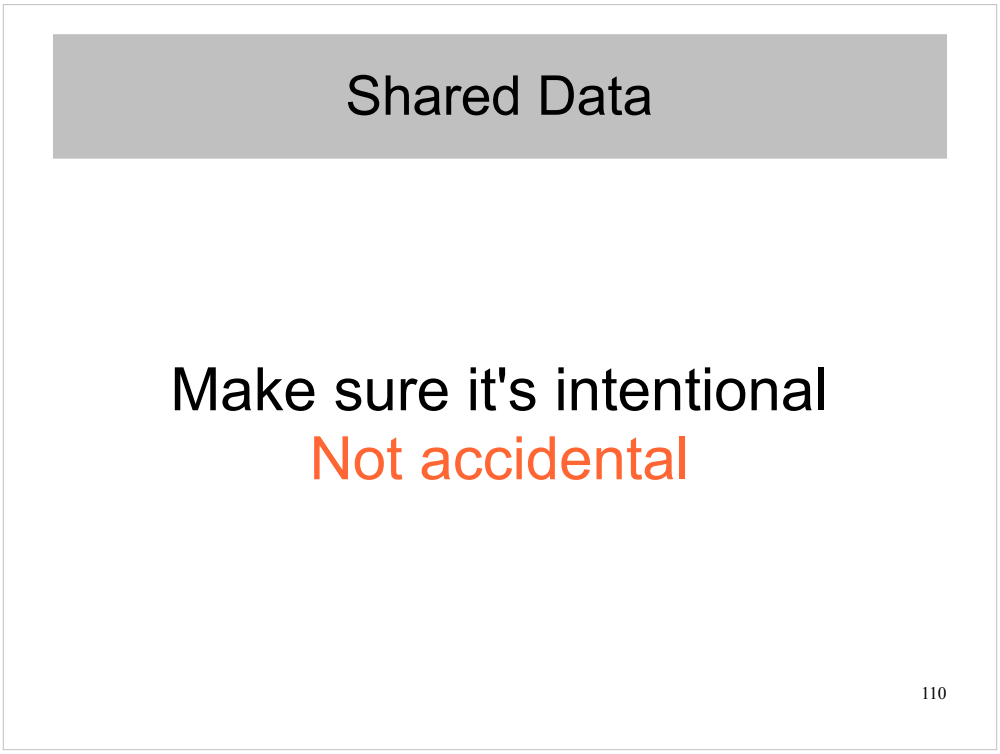
The other side of the coin is shared data.

Shared Data

Sometime available as static class variables.

Shared Data

Also available in closures,
for some uses



Changes in the prototype will be visible on the children.

Flyweight

Parser items,
Tick marks to a lesser extent.

Flyweight

Lots of copies that
vary in a few ways.

Near Miss

“I've got one that's almost right...”

For a time there was a case in Disk Clock where one disk was the same as another, except for the name.

Near Miss

Watch out for shared data -
Re-initialize all variable fields.

Sparse Objects

Large number of properties.

Most default.

Sparse Objects

Example:

Disks in Disk clock.

Most only need a few numbers,
But some override functions.

Nested Context

Like {block scope}.

(Or function scope in Javascript.)

Nested Context

```
new Module("dummy", function(mod){
  mod.require(...);
   mod.under('subdir', function(m) {
      m.require(...);
   });
});
```


In order to dodge the bullet of global state, I construct a module object to keep track of the relative path and other information.

Nested Context

```
new Module("dummy", function(mod){
  mod.require(...);
   mod.under('subdir', function(m) {
      m.require(...);
   });
});
```


I'm lazy.

I wanted a method to operate in a subdirectory, saving me from specifying the common path.

I could try to manage this by changing the path and then putting it back when done, but then I've got to worry about exceptions and other strange occurrences corrupting my state. It's enough to give a functional programer nightmares.

Nested Context

```
new Module("dummy", function(mod){
  mod.require(...);
   mod.under('subdir', function(m) {
      m.require(...);
   });
});
```


So I use a new object with a modified path.

You can probably guess by now how I do it...

Nested Context

```
under: function(path, f) {
  var m = Object.create(this);
  m.cd(path);
  f(m);
},
```


I call the object creator on the current object.

Nested Context

```
under: function(path, f) {
  var m = Object.create(this);
  m.cd(path);
  f(m);
},
```

Nested Context

```
under: function(path, f) {
  var m = Object.create(this);
  m.cd(path);
  f(m);
},
```

Patterns

Shared behavior

Shared data

Flyweights

Near-miss

Sparse objects

Nested Context

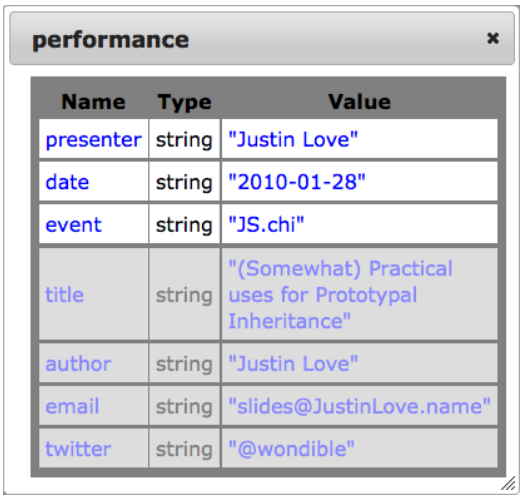
Resources

- http://javascript.crockford.com/prototypal.html
- http://bitbucket.org/JustinLove/
- http://JustinLove.name/presentations/protoint.pdf

- http://delicious.com/rauros/prototypal

- Contact me with questions.

var performance = Object.create(presentation);



performance

Name	Type	Value
presenter	string	"Justin Love"
date	string	"2010-01-28"
event	string	"JS.chi"
title	string	"(Somewhat) Practical uses for Prototypal Inheritance"
author	string	"Justin Love"
email	string	"slides@JustinLove.name"
twitter	string	"@wondible"

I hope you've enjoyed my performance.

Do we have time for questions?