software studio

Daniel Jackson

programming the browser

Java

› 1990: project at Sun to replace C++
› 1994: "Oak"retargeted to the web for "applets"
› Java takes off, first safe language in widespread use

Javascript

› 1995: "Mocha" project at Netscape
› Javascript takes off, included in Microsoft's IE
› 1996: submitted to Ecma as standard

today

› Java alive and well server-side
› but JS dominates client-side
› making inroads server-side too (eg, node.js)

syntax

statements like Java

› while, for, if, switch, try/catch, return, break, throw

comments

› use //, avoid /**/

semicolons

› inserted if omitted (yikes!)

declarations

› function scoping with **var**

functions

› are expressions; closures (yippee!)

```
var MAX = 10;
var line = function (i, x) {
    var l = i + " times " + x
          + " is " + (i * x);
    return l;
}
var table = function (x) {
    for (var i = 1; i <= MAX; i += 1) {
        console.log(line(i, x));
    }
}
// display times table for 3
table(3);
```

```
1 times 3 is 3
2 times 3 is 6
3 times 3 is 9
4 times 3 is 12
5 times 3 is 15
6 times 3 is 18
7 times 3 is 21
8 times 3 is 24
9 times 3 is 27
10 times 3 is 30
undefined
>
```

basic types

primitive types

› strings, numbers, booleans
› operators autoconvert

arrays

› can grow, and have holes

funny values

› undefined: lookup non-existent thing
› null: special return value

equality

› use ===, !==

```
> 1 + 2
3
> 1 + '2'
"12"
> 1 * 2
2
> 1 * '2'
2
```

```
> a = []
[]
> a[2] = 'hello'
"hello"
> a.length
3
> a[1]
undefined
> a[2]
"hello"
> a[3]
undefined
```

```
> 1 === 1
true
> 1.0 === 1
true
>'hello' === 'hello'
true
> [] === []
false
```

objects

literals

› *o = {prop: val, ...}*

properties

› get: *x = o.p*
› set, add: *o.p = e*
› delete: *delete o.p*

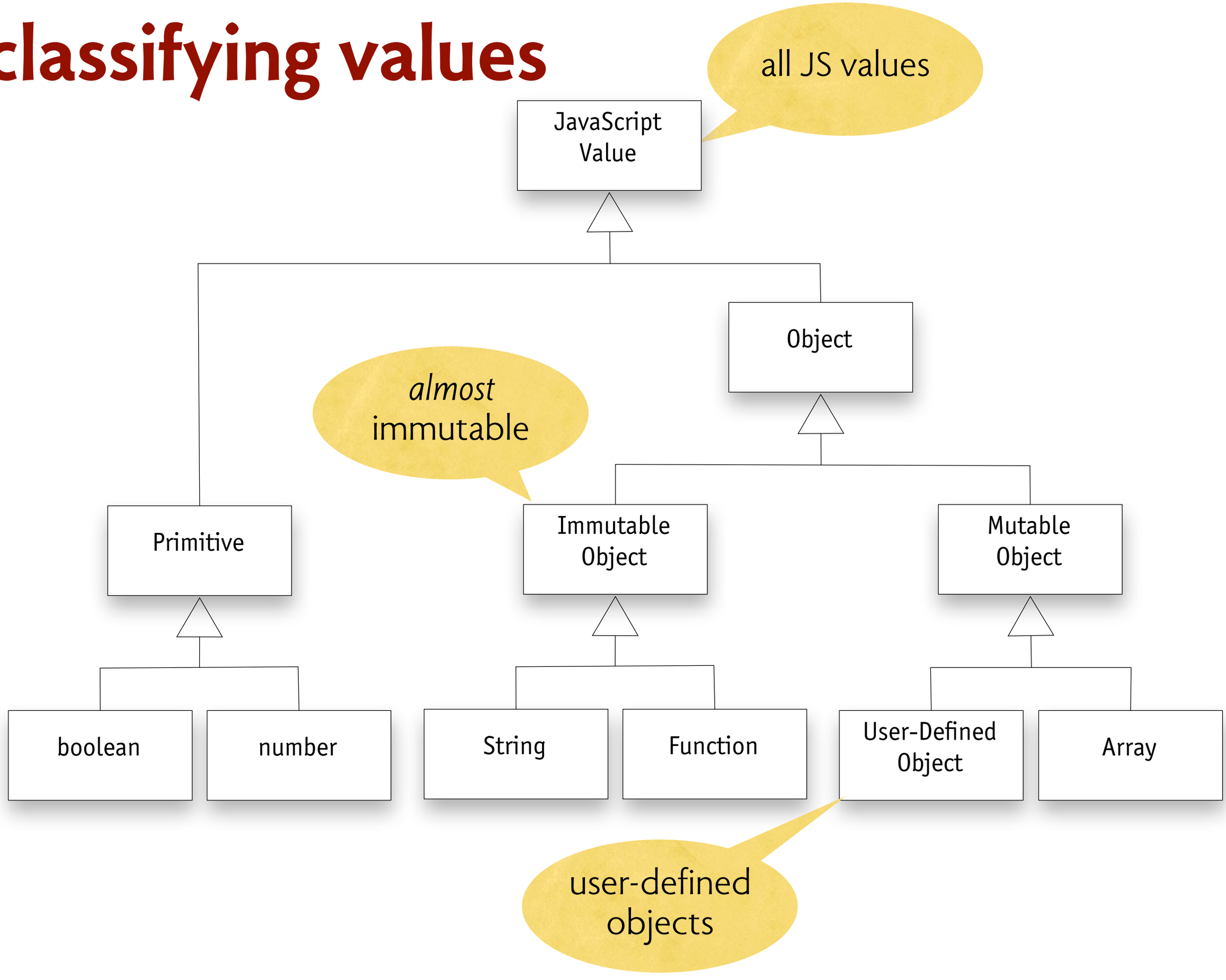
```
> point = {x: 1, y: 2}
Object
 1. x: 1
 2. y: 2
 3. __proto__: Object
> point.x
1
> point.z
undefined
> point.z = 3
3
> point.z
3
> delete point.z
true
> point.z
undefined
```

prototypes

› lookup along chain

```
> var Point = function (x, y) {this.x = x; this.y = y;}
undefined
> Point.prototype.magnitude = function () {return
Math.sqrt(this.x * this.x + this.y * this.y);}
function () {return Math.sqrt(this.x * this.x + this.y *
this.y);}
> p = new Point(1,2)
Point
> p.x
1
> p.magnitude
function () {return Math.sqrt(this.x * this.x + this.y *
this.y);}
> p.magnitude()
2.23606797749979
```

classifying values



typeof

```
> typeof(1)
"number"

> typeof(false)
"boolean"

> typeof('a')
"string"

> typeof([])
"object"

> typeof({})
"object"

> typeof(null)
"object"

> typeof(function () {})
"function"

> typeof(undefined)
"undefined"

> typeof(typeof(1))
"string"
```

note: no array ‘type’

note: no type type

aliasing & immutables

```
> x = ['h', 'i']
["h", "i"]

> y = x
["h", "i"]

> y[1] = 'o'
"o"

> x
["h", "o"]
```

aliasing

```
> x = ['h', 'i']
["h", "i"]

> x[1] = 'o'
"o"

> x
["h", "o"]

> x = 'hi'
"hi"

> x[1]
"i"

> x[1] = 'o'
"o"

> x[1]
"i"
```

some objects
can't be modified

immutables are our friends!

no mention of x
⇒ no change to x

equality and truthiness

like many scripting languages

› features that save 3 seconds of typing
› instead cost 3 hours of debugging

examples

› behavior of built-in equality operator ==
› strange rules of “truthiness”

```
> 0 == ''
  true
> 0 == '0'
  true
> '' == '0'
  false
```

```
> 0 === ''
  false
> 0 === '0'
  false
> '' === '0'
  false
```

```
> (false) ? "true" : "false";
"false"
> (null) ? "true" : "false";
"false"
> (undefined) ? "true" : "false";
"false"
> (0) ? "true" : "false";
"false"
> ('') ? "true" : "false";
"false"
> (undefined == null)
true
> (undefined === null)
false
```

numbers & non-numbers

good news

- no int vs float
- exponents

bad news

- strange NaN value
- no infinite precision

puzzle

- is this good practice?

```
if (f() != NaN) {...}
```

- no, need

```
if (isNaN(f())) {...}
```

```
> 1 + 2
3
> 1.0 + 2
3
> 1/3
0.3333333333333333
> 1000000 * 1000000
1000000000000
> 1000000 * 1000000 * 1000000
1000000000000000000
> 1000000 * 1000000 * 1000000 * 1000000
1e+24
```

```
> 1/0
Infinity
> (1/0) * 0
NaN
```

```
> typeof(NaN)
"number"
> NaN === NaN
false
> NaN !== NaN
true
```

undefined, reference error & null

notions of bad access

› ReferenceError exception
› undefined value (built-in)
› null (predefined value)

how to use them

› ReferenceError: regard as failure if raised
› null: best left unused, IMO
› undefined: unavoidable (eg, optional args)

a little paradox

› undefined is a defined value for a variable

```
> newvar
ReferenceError: newvar is not defined
> newvar = undefined
undefined
> newvar
undefined
> obj = {}
Object
> obj.f
undefined
> obj.f = null
null
```

lookup

to evaluate an expression

› lookup value of each var

› apply functions to arguments

how to lookup

› just find the binding for the var

```
> h = "hello there"
"hello there"
> escape
function escape()
{ [native code] }
> escape(h)
"hello%20there"
```

assignment

assignment statement

› x = e, read “x gets e”

semantics

› evaluate e to value v

› if x is bound, replace value with v

› else create new binding of x to v

in JS, all names are vars

› a function name is just a var, can reassign

› more on this when we see recursion

contrast to Java

› variables just one kind of name

› other kinds of name: methods, classes, packages

```
> h = "hello there"
"hello there"
> escape(h)
"hello%20there"
> escape = function()
{return "gone!";}
function () {return
"gone!";}
> escape(h)
"gone!"
```

aliasing

after the assignment x = y

› x is bound to same value as y

how sharing arises

› no implicit copying

› so x and y are names for same object

consequence

› change to "one" affects the "other"

if object is immutable

› no change to object possible

› so as if value is copied

```
> y = []
[]
> x = y
[]
> x.f = 1
1
> y.f
1
```

making functions

function expression

› **function** (args) {body}

functions are 'polymorphic'

› implicit typed

› depends on how args used

```
> three = function () {return 3;}
function () {return 3;}
> three
function () {return 3;}
> three()
3
> id = function (x) {return x;}
function (x) {return x;}
> id(3)
3
> id(true)
true
> id(id)
function (x) {return x;}
> (id(id))(3)
3
```

functions are first class

just like other objects

- can bind to variables
- can put in property slots
- can add property slots

```
> seq = function () {
      seq.c += 1; return seq.c;}
function () {seq.c += 1; return
seq.c;}
> seq.c = 0
0
> seq()
1
> seq()
2
```

note: bad lack of encapsulation! will fix later with closures

```
> seq = function () {return (seq.c = seq.next(seq.c));}
function () {return (seq.c = seq.next(seq.c));}
> seq.c = 0
0
> seq.next = function (i) {return i + 2;}
function (i) {return i + 2;}
> seq()
2
> seq()
4
```

two phases

```
> (function (x) {return x + 1;}) (3)
4
```

creation

› function expression evaluated

application

› function body evaluated

evaluation order for applications

› first evaluate arguments, left to right

› then evaluate body

```
> log = function (s) {console.log(s + seq());}
function (s) {console.log(s + seq());}
> (function () {log('c')}) (log('a'),log('b'))
a1
b2
c3
```

evaluating the body

what environment is body evaluated in?

› same environment application is evaluated in?

let's see!

› hmm...

```
> x = 1
1
> f = (function (x) {return function () {return x;};}) (x)
function () {return x;}
> f()
1
> x = 2
2
> f()
1
```

two environments

when function is created

› keeps environment as a property
› called ‘function scope’
› uses this environment to evaluate body in

what about arguments?

› new environment (‘frame’) with bindings for args
› linked to function scope

example 1

```
> f = function () {return x;}
function () {return x;}
> x = 1
1
> f()
1
> x = 2
2
> f()
2
```

what happens here?

- function scope is top-level environment
- assignment to x modifies binding in top-level environment
- so in this case x refers to x of application environment too

example 2

```
> f = function (x) {return x;}
function (x) {return x;}
> x = 1
1
> y = 2
2
> f(y)
2
```

what happens here?

- function scope is top-level environment
- when application is evaluated, argument x is bound to 2
- local x said to shadow global x

example 3

```
> x = 1
1
> f = (function (x) {return function () {return x;};}) (x)
function () {return x;}
> f()
1
> x = 2
2
> f()
1
```

what happens here?

- when f is applied, x is bound to 1 in new frame
- anonymous function has scope with x bound to 1
- assignment to top-level x does not modify this scope

example 4

```
> f = (function (x) {return function () {x += 1; return x;};}) (0)
function () {x += 1; return x;}
> f()
1
> f()
2
```

what if we modify x?

- when f is applied, x is bound to 0 in new frame
- anonymous function has scope with x bound to 0
- this ‘internal’ x is updated every time f is called

arrays: a refresher

Javascript operations

- push/pop (back)
- unshift/shift (front)
- splicing
- concatenation

autofilling

- if set at index beyond length
- elements in between set to undefined

```
> a = [3,5,7]
[3, 5, 7]
> a.push(9)
4
> a
[3, 5, 7, 9]
> a.unshift(1)
5
> a
[1, 3, 5, 7, 9]
> a.pop()
9
> a
[1, 3, 5, 7]
> a.shift()
1
> a
[3, 5, 7]
> a.splice(1,1,6)
[5]
> a
[3, 6, 7]
> a[4] = 8
8
> a
[3, 6, 7, undefined, 8]
```

map

```
map = function (a, f) {
    var result = [];
    each (a, function (e) {
        result.push(f(e));
    });
    return result;
}
```

type

› map: list[A] x (A→B) → list[B]

```
> twice = function (x) {return x * 2;}
function (x) {return x * 2;}
> a = [1,2,3]
[1, 2, 3]
> map (a, twice)
[2, 4, 6]
```

fold (or reduce)

```
fold = function (a, f, base) {
    var result = base;
    each (a, function (e) {
        result = f(e, result);
    });
    return result;
}
```

type

› fold: list[A] x (A x B→B) x B → B

```
> times = function (x, y) {return x * y;}
function (x, y) {return x * y;}
> a = [1,2,3]
[1, 2, 3]
> reduce (a, times, 1)
6
```

filter

```
filter = function (a, p) {
    var result = [];
    each (function (e) {
        if (p(e)) result.push(e);
    });
    return result;
}
```

type

› filter: list[A] x (A→Bool) → list[A]

```
> a = [1, 3, 5]
[1, 3, 5]
> filter (a, function (e) {return e < 4; })
[1, 3]
```

6.170 Software Studio
Spring 2013