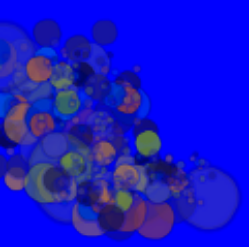


JavaScript objects & functions

Why JavaScript?

It adds a scripting language to your armoury

It is the one you are least likely to be able to avoid

The language has been improved (strict mode)

There have been advances in speed (V8)

It has moved off the Web (node.js)

What kind of JavaScript?

It is said that people ***use*** JavaScript without ***learning*** it

We will learn it ***away from browsers***, which will avoid lots of complications

We will use ***node.js***, which has its own module system, and its own libraries, not like the browser ones

We will use the ***strict mode*** of JavaScript 5, which gets rid of some obsolete rubbish

Resources

Online books: Eloquent JavaScript and wikibooks

Quick syntax reference: wikipedia

Node library reference: nodejs.org/api/

Node module system: nodejs.org/api/modules.html

The language standard for JavaScript 5: ECMA

There are lots of other references you can Google (e.g. JSLint, QUnit): pick ones which aren't related to browsers, and beware differences in strict mode, the module system, and the node.js libraries

Node.js at home

You can install node.js yourself from nodejs.org

You will get a command *node* (or possibly *nodejs* or *js*)

You should also get a command *npm* (node package manager) which you can use to install extra packages


KEEP
CALM
AND
USE
STRICT

Learning by doing


```
var answer = 41;
anwser = answer + 1;
console.log(answer);
```

```
js> node {f}
41
js>
```

Leniency

Old JavaScript is too lenient, for example a miss-spelling leads to a program that works but gives wrong answers

This leniency comes from the idea that JavaScript should 'just work' for non-experts

The idea is no good if (a) it leads to bad programs or (b) the rules are too complex or (c) debugging is too difficult

JavaScript 5 has a ***strict mode*** which switches off bad effects (while staying compatible with old interpreters)

Strict mode


```
"use strict";
var answer = 41;
anwser = answer + 1;
console.log(answer);
```

```
js> node {f}
.../{f}:3
anwser = answer + 1;
         ^
ReferenceError: anwser is not defined
js>
```

Being strict

Advice: put "use strict" at the top of every program or module, otherwise debugging will be much harder

The opposite of ***strict mode*** is called ***sloppy mode***

Old interpreters ignore "use strict", which is legal but does nothing, and stay in sloppy mode

Advice: use semicolons, otherwise your editor will probably mess up the indenting

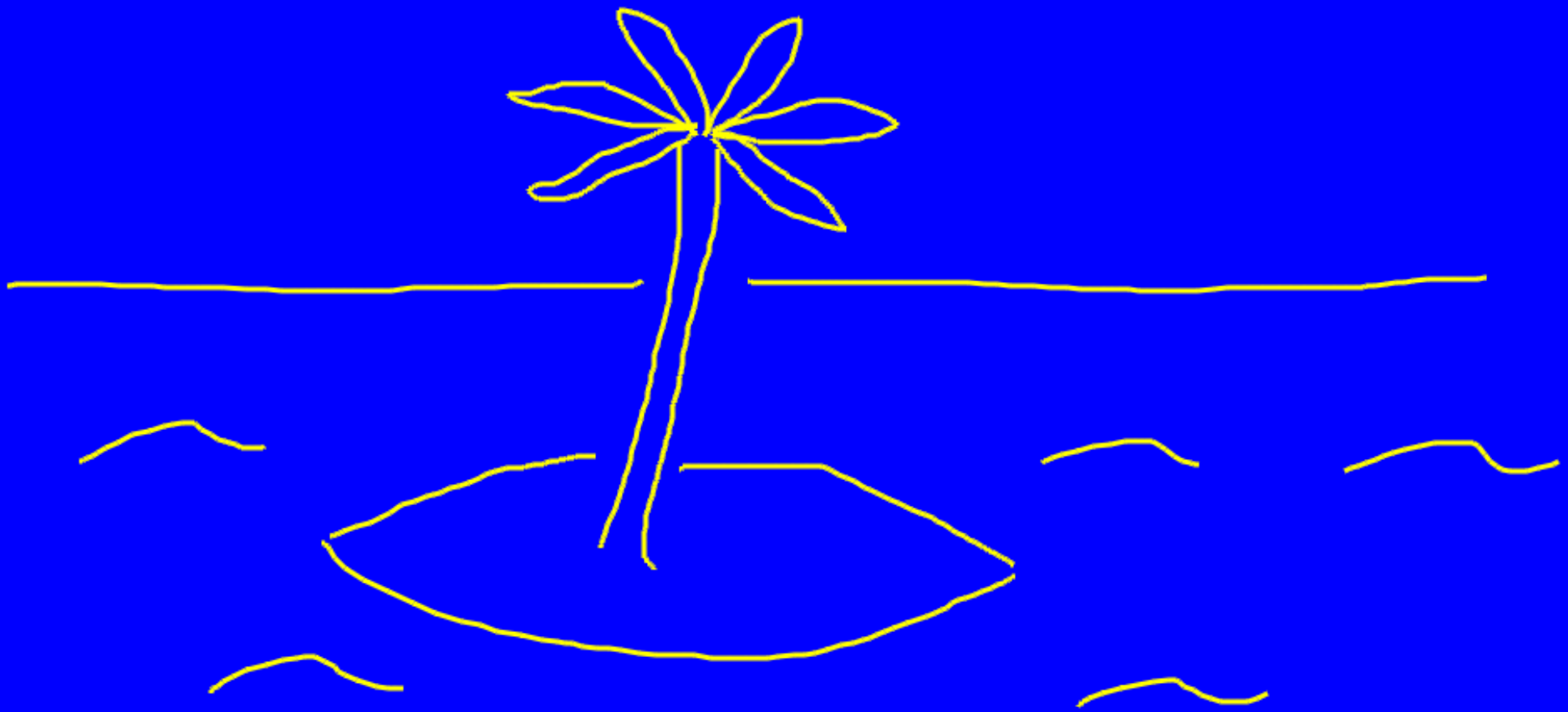
Target program

When programming experimentally, it is useful to have a target program to work towards

Our target program is going to be an adventure game

To begin with, it is all about objects

Setting



Playing the game


```
js> node {f}
You are near a tree
There is a spade here
> go south
You are on a beach
There is sand here
> dig sand
You have no spade
> go north
You are near a tree
There is a spade here
> take spade
You are carrying a spade
> go south
You are on a beach
There is sand here
> dig sand
You find your holiday return ticket
and go home
js>
```

Tree Object


```
"use strict";
var tree = { south: beach };
```

```
js> node {f}
.../{f}:2
var tree = { south: beach };
                    ^
ReferenceError: beach is not defined
js>
```

Simple objects

A simple object can be created just by defining its fields in curly brackets

The *:* character after a field name ***declares*** the field, just as ***var*** before a variable declares the variable

The objects which fields refer to must exist

The beach object


```
"use strict";
var tree = { south: beach };
var beach = { north: tree };
console.log(tree);
```

```
js> node {f}
{ south: undefined }
js>
```

Two phases

The JavaScript system has two phases:

One it collects ('lifts') variable names (and functions)

Two it executes code (including initialising variables)

When `beach` didn't exist, the system should have said "not declared" instead of "not defined", which looks a lot like "undefined" – this is probably a historical accident

The C family

These should be familiar:

```
if (t) { s; }
if (t) { s; } else { t; }
t ? v : w
switch (e) { case c: s; break; }
for (i=0; i<n; i++) { s; }
while (t) { s; }
do { s; } while (t);
break; continue;
```

Loopy


```
"use strict";
var grid = [["tl","tm","tr"], ["bl","bm","br"]];
for (var i=0; i<2; i++) {
  var row = grid[i];
  for (var i=0; i<3; i++) {
    console.log(row[i]);
  }
}
```

```
js> node {f}
tm
tl
tr
js>
```

'Local' variables

When you write `for (var i=0; ...)` then `var i` ***looks like*** a local variable, but it isn't

It gets lifted to the top of the program (or function)

So, two nested `var i` declarations give the same `i`

Duplicate declarations are probably allowed because of the idiom:

```
if (t) var i=0; else var i=1;
```

Check beach


```
"use strict";
var tree = { south: beach };
var beach = { north: tree };
console.log(tree);
console.log(beach);
```

```
js> node {f}
{ south: undefined }
{ north: { south: undefined } }
js>
```

Objects point to each other


```
"use strict";
var tree = {}, beach = {};
tree.south = beach, beach.north = tree;
console.log(tree);
console.log(beach);
```

```
js> node {f}
{ south: { north: [Circular] } }
{ north: { south: [Circular] } }
js>
```

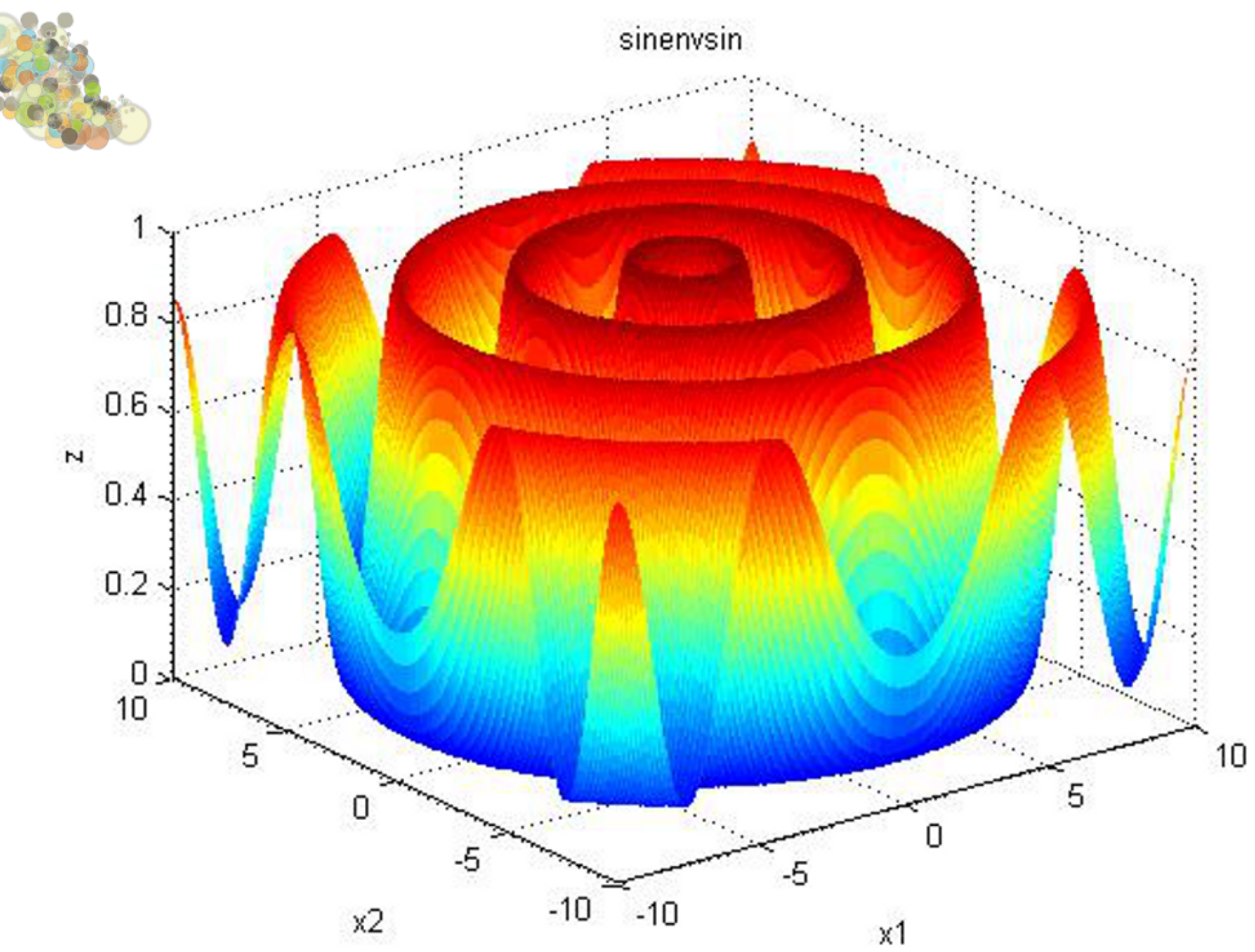
Updating objects

Although printing doesn't involve variable names, only values, it does detect cycles to prevent an infinite loop

You can update an object at any time, by adding or replacing or deleting a field:

- `tree.south = x;`
- `tree.south = y;`
- `delete tree.south;`

Deleting fields is an efficiency trap in modern JavaScripts; it is better to avoid doing it


sinenvsin
1
0.8
0.6
0.4
0.2
0
z
10
5
0
-5
-10
x2
-10
-5
0
5
10
x1


I'D LIKE A PLATE OF CURRIED FUNCTIONS... AND GO EASY ON THE SYNTACTIC SUGAR...
?
I'M NEVER GOING ANYWHERE WITH HIM AGAIN!

A function


```
"use strict";

function go() {
  console.log("going");
}

console.log(go);
```

```
js> node {f}
[Function: go]
js>
```

Calling a function


```
"use strict";

function go() {
  console.log("going");
}

go();
```

```
js> node {f}
going
js>
```

Forward calling


```
"use strict";

go();

function go() {
  console.log("going");
}
```

```
js> node {f}
going
js>
```

Forward call and reference


```
"use strict";

go();

function go() {
  console.log(s);
}

var s = "going";
```

```
js> node {f}
undefined
js>
```

Forward reference and call


```
"use strict";

function go() {
  console.log(s);
}

var s = "going";

go();
```

```
js> node {f}
going
js>
```

Compiling

The compiling process is (a) lift the variables (b) lift the functions (c) create a main function from the rest

The result of translating to C might be:

```c
object *s;

void go() { printf(..., s); }

int main() {
  s = string("going");
  go();
}
```

Attaching functions


```
"use strict";
var tree = {};
tree.go = go;

function go() {
  console.log("going");
}
console.log(tree);
```

```
js> node {f}
{ go: [Function: go] }
js>
```

Functions as objects

A function is an object

A function can be attached as a field

A function can be passed as an argument or returned as a result

If you call a function with too few args, the rest are `undefined`

If you call a function with too many args, the rest are ignored

Call attached function


```
"use strict";
var tree = {};
tree.go = go;

function go() {
  console.log("going");
}

tree.go();
```

```
js> node {f}
going
js>
```

Naming


```
"use strict";
var tree = {};
tree.go = fun;

function fun() {
  console.log("going");
}

tree.go();
```

```
js> node {f}
going
js>
```

Call method


```
"use strict";
var tree = {};
tree.where = "near a tree";
tree.go = go;

function go() {
  console.log("Go", this.where);
}

tree.go();
```

```
js> node {f}
Go near a tree
js>
```

Methods

When you call a function which is attached to an object, the variable ***this*** gets set to the object during the call

A function attached to an object is called a ***method***

The function can still be called directly, but:

- in strict mode, ;***this*** is set to `undefined`
- in sloppy mode, ***this*** is set to the global object

Field details


```
"use strict";
var tree = { "where is": "near a tree" };
console.log(tree["where is"]);

var beach = {};
beach["where is"] = "on a beach";
console.log(beach["where is"]);

var x = "where is";
console.log(tree[x]);
```

```
js> node {f}
near a tree
on a beach
near a tree
js>
```

Dynamic fields

`tree["where"]` is the same as `tree.where`

It allows:

- non-identifier field names: `tree["where is"]`
- run-time field names: `tree[x]`
- an object to act like a hash table with string keys

Call method in two steps


```
"use strict";
var tree = { go:go, where:"near a tree" };

function go() {
  console.log("Go", this.where);
}
var fun = tree.go;
fun();
```

```
js> node {f}
.../{f}:5
  console.log("Go", this.where);
                          ^
TypeError: Cannot read property
'where' of undefined
js>
```

Pick a method and call it

You can't split up `tree.go()` into two separate steps, pick `fun=tree.go` and then call it with `fun()`

JavaScript notices the two together and adds the extra step of setting `this` to `tree`

Call a run-time method


```
"use strict";
var tree = { go:go, where:"near a tree" };

function go() {
  console.log("Go", this.where);
}

tree["go"]();
```

```
js> node {f}
Go near a tree
js>
```

Dynamic method choice

It seems that *node* does the right thing with `tree["go"]()`, but that is a recent change

Does every Javascript system do the same? You can't rely on it.

Is there another way? Yes.

Calling call


```
"use strict";
var tree = { go:go, where:"near a tree" };

function go() {
  console.log("Go", this.where);
}

var f = tree.go;
f.call(tree);
```

```
js> node {f}
Go near a tree
js>
```

The call method

Every function has a `call` method attached

The expression `f.call(obj, x, y)` is equivalent to `obj.f(x,y)` except that function `f` doesn't have to be attached to `obj`

It sets `this` equal to `obj` and passes the remainder as arguments

Summary

- Objects have fields, you can add fields at will
- Functions are objects
- A function-field is a method
- A *property* is a field or a method
- You can set `obj = {f1:v1,"f2":v2,...}`
- `obj.f` gets a field, `obj[x]` looks it up
- `obj.f(...)` calls a method
- `f.call(obj,...)` is a non-attached equivalent
- `obj[x].call(obj,...)` calls a named method