The JavaScript Language (Part 3)

Robert M. Dondero, Ph.D.
Princeton University

Objectives

- We will cover:
  - A subset of JavaScript…
  - That is appropriate for COS 333…
  - Through example programs

Agenda

- **Prototypes**
- Delegation to prototypes
- Classes

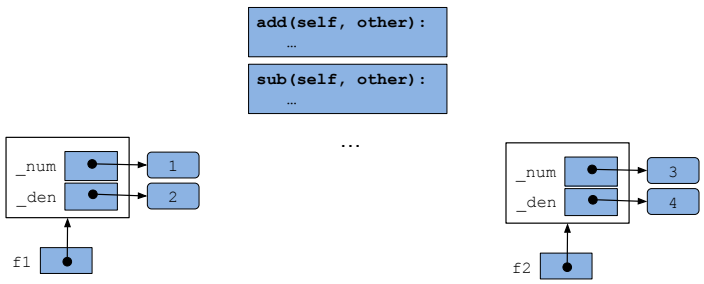
Prototypes

- Recall **fraction2.js**, **fraction2client.js**…
- **Problem**:
  - Space inefficiency…

Prototypes

Python

```
f1 = Fraction(1, 2)
f2 = Fraction(3, 4)
```

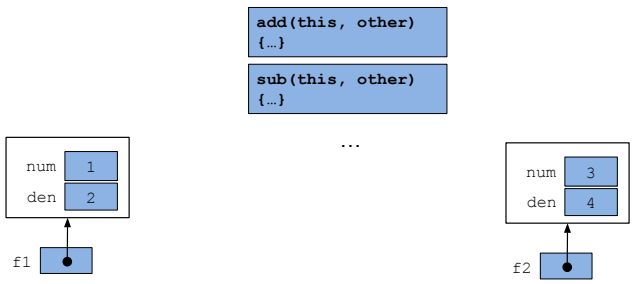


Explicit `self` parameter allows `Fraction` objects to share same function defs

Prototypes

Java

```
Fraction f1 = new Fraction(1, 2);
Fraction f2 = new Fraction(3, 4);
```

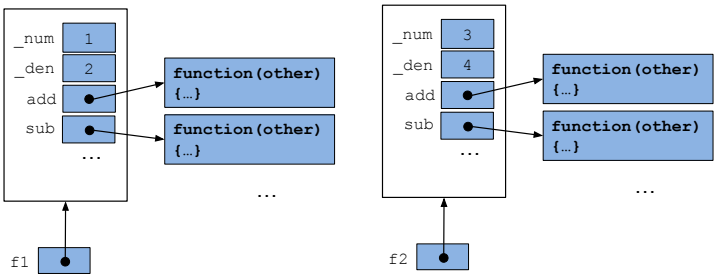


Implicit `this` parameter allows `Fraction` objects to share same method defs

Prototypes

JavaScript (so far)

```
let f1 = createFraction(1, 2);
let f2 = createFraction(3, 4);
```



Each fraction object has its own set of function defs
Many fraction objects => massive space inefficiency

Prototypes

- **Problem**:
  - Space inefficiency
- **Solution (part 1)**:
  - Prototypes

Prototypes

- See **fraction3.js**, **fraction3client.js**
  - The job:
    - Same as fraction2.js and fraction2client.js

Prototypes

[see slide]

Code notes

Fraction() is a ***constructor function***

Designed to be called via ***new operator***

Similar to createFraction(), but…

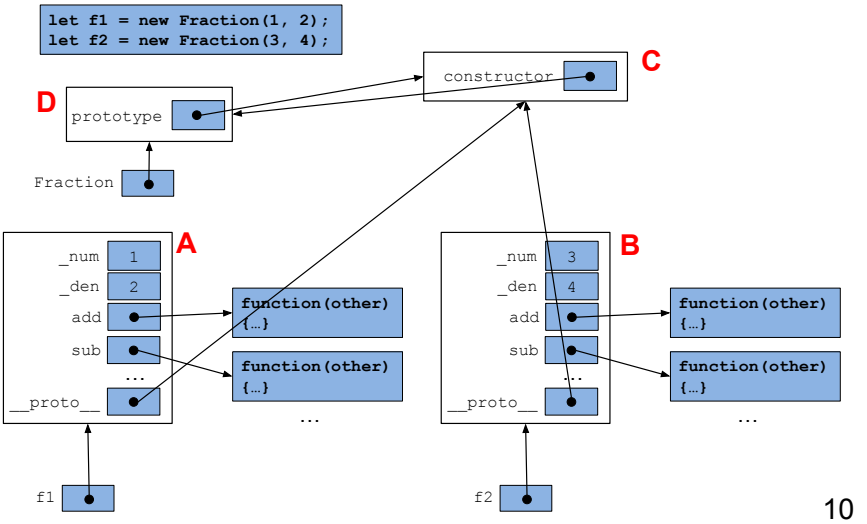
Automatically creates new object

Automatically returns new object

Calling a constructor function creates a ***prototype***

Prototypes

JavaScript (so far)



Agenda

- Prototypes
- **Delegation to prototypes**
- Classes

Prototypes

- **Problem**:
  - Space inefficiency
- **Solution (part 1)**:
  - Prototypes
- **Solution (part 2)**:
  - Delegation to prototypes

Delegation to Prototypes

- See **fraction4.js**, **fraction4client.js**
  - The job:
    - Same as fraction3.js and fraction3client.js

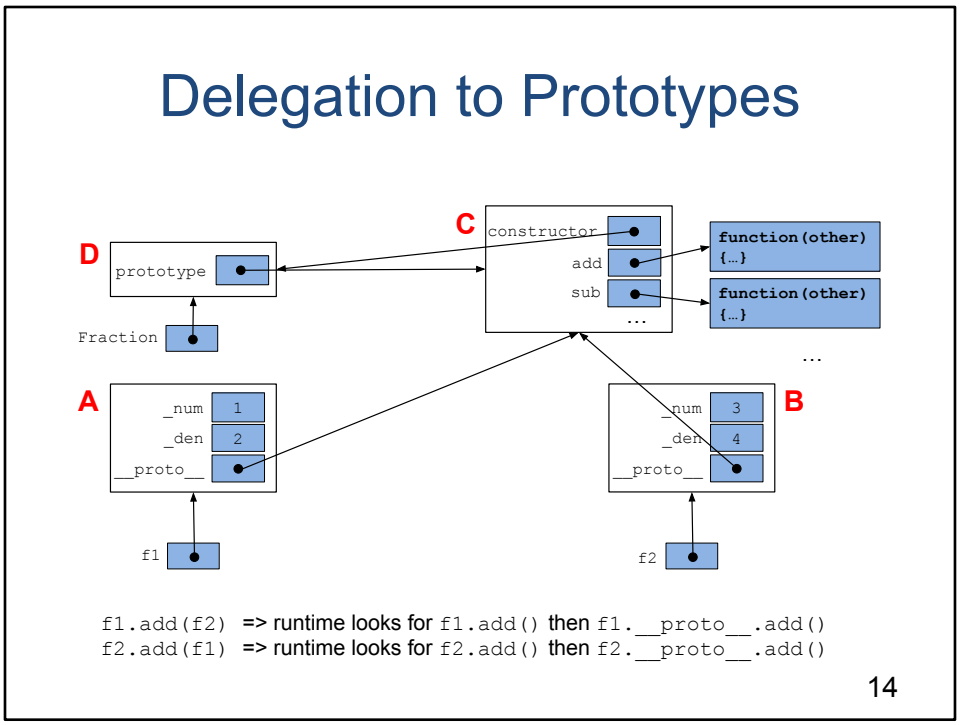
13

Delegation to Prototypes

[see slide]

Code notes:

Stores functions in Fraction.prototype object



Delegation to Prototypes

[see slide]

This is the structure of fraction4.js and fraction4client.js

The last piece of the puzzle: **delegation**

Each Fraction object automatically **delegates** to its prototype

Each Fraction object automatically delegates work to Fraction.prototype (the object labeled C)

Each Fraction object automatically delegates work to the object referenced by its __proto__ property

f1.add(f2)

Runtime first looks for add() in f1 (the object labeled A)

Runtime then looks for add() in f1's prototype (the f1._proto_ object) (the object labeled C)

That solves the space inefficiency problem

Agenda

- Prototypes
- Delegation to prototypes
- **Classes**

Classes

- **Problem**
  - Delegation to prototypes is distant from mainstream OOP
  - Difficult to learn & understand
- **Solution**
  - “Class” syntax in ES6

Classes

Problem

Delegation to prototypes is distant from mainstream OOP

As implemented in C++, Java, Python, …

Difficult to learn & understand

Solution

“Class” syntax in ES6

Classes

- See **fraction5.js**, **fraction5client.js**
  - The job:
    - Same as fraction4.js and fraction4client.js

17

Classes

[see slide]

Code notes

fraction5client.js is identical to fraction4client.js
fraction5.js uses ES6 “class” syntax
fraction5.js transpiles exactly to fraction4.js

Classes

- JavaScript really doesn't have:
  - Classes
  - Objects as instances of classes
- JavaScript has:
  - Objects
  - Delegation to prototypes

18

Classes

[see slide]

Can't get away with knowing only the ES6 syntax
Must know the underlying mechanisms

Aside: Prototype Chains

- JavaScript really doesn't have:
  - Inheritance
- JavaScript has:
  - Prototype chains
  - (Beyond our scope)

Aside: this

- **Question**: How is `this` bound within a function `f()`?
- **Answer**: Depends upon how `f()` is called:

Function Call	Binding of `this`
`f()`	In `f()`, `this` is `undefined`
`obj.f()`	In `f()`, `this` is bound to `obj`
`new f()`	In `f()`, `this` is bound to a new empty object
`f.call(obj)`	In `f()`, `this` is bound to `obj`

JavaScript Commentary

- **Evolutionary path 1**
  - Classes
- **Evolutionary path 2**
  - Delegation to prototypes

- Path 1 won
  - But JavaScript survived!

JavaScript Commentary

Evolutionary path 1
Classes
Early languages: Simula, Smalltalk
Later languages: C++, Java, Python, …

Evolutionary path 2
Delegation to prototypes
Early language: Self
Later languages: JavaScript, TypeScript

Path 1 won
But JavaScript survived!

Summary

- We have covered:
  - Prototypes
  - Delegation to prototypes
  - Classes