

# CS450

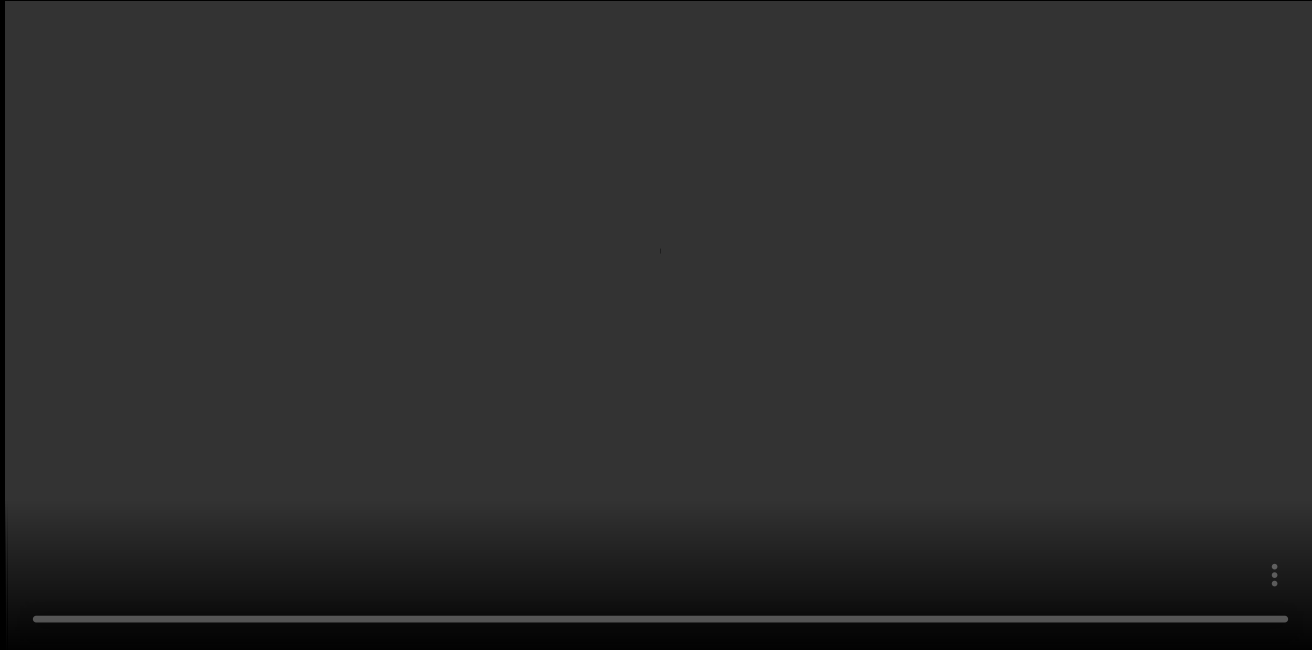
## Structure of Higher Level Languages

Lecture 24: JavaScript intro; LambdaJS intro

Tiago Cogumbreiro

Today we will learn...

Hint: `(!+[]+[]+![]).length == 9`



Source: Gary Bernhardt, 2012. [www.destroyallsoftware.com/talks/wat](http://www.destroyallsoftware.com/talks/wat)

# In this module we will...

- Introduce Object-Oriented Programming (OOP)
- Introduce "The Essence of Javascript"
- Implement JavaScript in Racket

# Why JavaScript?

- JavaScript is everywhere
- JavaScript's object system is simple to describe
- JavaScript introduces a minimal yet powerful OOP model

## Technically, ECMAScript

However, the umbrella term "JavaScript" as understood in a web browser context contains several very different elements. One of them is the core language (ECMAScript), another is the collection of the Web APIs, including the DOM (Document Object Model).

Source: [MDN](#)

# What are we learning?

## The *language mechanics* of JavaScript

### JavaScript **semantics**

1. What is an object?
2. How does variable binding work?
3. How does inheritance work?
4. How does mutation work?
5. How do functions interact with objects?

# What are we **not** covering?

1. JavaScript API's (including the standard library)
2. promises/async/await
3. Control flow (for, while, if, can be represented as  $\lambda$ -calculus)
4. JavaScript best practices
5. Differences between ECMAScript versions
6. A faithful implementation of ECMAScript (simplicity trumps fidelity)

# How are we learning JavaScript?

1. We will study a research paper that introduces the core JavaScript semantics
2. We will implement these semantics in Racket

## The Essence of JavaScript

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi

Brown University

**Abstract.** We reduce JavaScript to a core calculus structured as a small-step operational semantics. We present several peculiarities of the language and show that our calculus models them. We explicate the desugaring process that turns JavaScript programs into ones in the core. We demonstrate faithfulness to JavaScript using real-world test suites. Finally, we illustrate utility by defining a security property, implementing it as a type system on the core, and extending it to the full language.

[Essence of JavaScript. Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. ECOOP. 2010.](#)





# $\lambda_{JS}$ : the essence of JavaScript

## Paper summary

1. **Describes a core language:**  $\lambda_{JS}$  is a  $\lambda$ -calculus extended with shared memory (references), and with an object system
2. **Describes a translation function:** translates JavaScript into  $\lambda_{JS}$

## Our study

- We will learn how to implement the object system of JavaScript
- Our final project consists of **implementing a translation function from SimpleJS into  $\lambda_{JS}$**

Paper URL: [cs.brown.edu/research/pl1t/d1/jssem/v1/gsk-essence-javascript-r6.pdf](https://cs.brown.edu/research/pl1t/d1/jssem/v1/gsk-essence-javascript-r6.pdf)



# Some definitions on translators

- **Translator** (or translation function): the process of converting terms of one language into terms of another language (possibly the same language)
- **Compiler**: translate a source language into a target language; generally the target language is at the machine-level (e.g., assembly, bytecode, intermediate-representation)
- **Syntactic desugaring**: the process of converting syntactic abbreviations into more fundamental terms of the same language
- **Source-to-source translator**: unlike the compiler, a source-to-source compiler, the target language of a source-to-source translator is a higher-level language
- **Transpiler**: source-to-source translator. Term popularized by the JavaScript community.
- **Polyfill**: provides modern functionality on older browsers (eg, via syntactic desugaring).

# Let us learn JavaScript and $\lambda_{JS}$

Following, I will:

1. introduce the basics of JavaScript
2. introduces the basics of  $\lambda_{JS}$
3. relate the functionalities found in JavaScript to those of  $\lambda_{JS}$
4. list our  $\lambda_{JS}$  AST written in Racket

# Constants

# Constants

$c ::= n \mid s \mid b \mid \text{undef}$

```
// Numbers
100
-100
// Strings
"foo"
'foo'
// Booleans
true
false
// Undefined
undefined
```

# Expressions in JavaScript

# Expressions

1. Objects
2. Functions

■ In JavaScript functions are also objects! We will learn

# Objects in JavaScript



# Objects are mutable maps

1. **Object declaration:** `{"str1" : expr1, "str2": expr2, ... }`
2. **Field lookup:** `obj["field"]` or `obj.field`
3. **Field update:** `obj["field"] = expr` or `obj.field = expr`
4. **Field deletion:** `delete obj["field"]` or `delete obj.field`

```
var x = {} // Creates a new, empty object
// Reading an unset field returns undefined
console.assert(x["foo"] == undefined)
// Assignment works as expected
x["foo"] = 20
console.assert(x["foo"] == 20);
// Deleting a field is equivalent to assigning to undefined
delete x["foo"]; // x["foo"] = undefined ← in our semantics
console.assert(x["foo"] == undefined);
```

What is the difference between `var`, `let`, and `const`?

# var variable declaration

`var` declares a function-global variable that can be assigned.

```
var x = 1;

if (x == 1) {
  var x = 2; // We can redeclare the function-global x in any scope
  console.assert(x == 2);
}

console.assert(x == 2);
x = 10; // We can safely assign to x
console.assert(x == 10);
```

Source: [MDN](#)

# let variable declaration

`let` creates a local variable. `let` cannot be redeclared in the same scope, but can be redeclared in other scopes. A variable declared with `let` **can** be assigned. Source: [MDN](#)

```
let x = 1;

if (x === 1) {
  let x = 2; // A new scope declares a new variable x
  console.assert(x === 2);
}
// let x = 2; // Expected: SyntaxError
console.assert(x === 1);
x = 10; // We can safely assign a new value to x
console.assert(x === 10);
```

# const variable declaration

`const` creates a local variable. `const` cannot be redeclared in the same scope, but can be redlared in other scopes. A variable declared with `let` **cannot** be assigned. Source: [MDN](#)

```
const number = 42;
{ const number = 52; } // each block creates a new scope
try {
  number = 99;
  console.assert(false);
} catch(err) { console.log(err); } // expected output: TypeError
// const number = 99; // expected output: SyntaxError

console.assert(number == 42);
```

# JavaScript and creating objects

```
{ } // The empty object
> { "a key": 100, "another key": true, "": null, "00": undefined }
{ 'a key': 100, 'another key': true, '': null, '00': undefined }
> { 100: "numbers are converted to strings", true: "booleans are left as is",
  undefined: "as are undefined", null: "and also null" }
{ '100': 'numbers are converted to strings',
  true: 'booleans are left as is', undefined: 'as are undefined', null: 'and also null' }
> { "foo": 1, "foo": 2 + 100 } // Only the left-most key sets its value
{ foo: 102 } // Evaluation works on values (but not on not keys)
```

■ In our implementation, we will assume the keys can **only** be strings

# Functions in JavaScript

# JavaScript and functions

## Nameless functions

We can write lambda abstraction in JavaScript with the following syntax

$$(x \dots) \Rightarrow e$$

where  $x \dots$  is a comma-separated list of identifiers.

### Example in JS

```
console.assert(  
  ((x)  $\Rightarrow$   
    (y)  $\Rightarrow$  x - y)(1)(2)  
  ==  
  -1);
```

### Racket

```
(check-equals?  
  (((lambda (x)  
    (lambda (y) (- x y)))) 1) 2)  
  -1)
```



# Named functions in JavaScript

Named functions in JavaScript have the following syntax.

```
function x(x...) { stmt ... }
```

JavaScript

```
function add(x, y) {  
  return x + y;  
}  
console.assert(add(1, 2) == 3);
```

Racket

```
(define (add x y)  
  (+ x y))  
(check-equals? (add 1 2) 3)
```

# Syntactic sugar

## Version 1

```
p1 = {};  
p1["x"] = 13  
p1["y"] = 7;  
console.assert(p1["x"] == 13);  
console.assert(p1["y"] == 7);
```

# Syntactic sugar

## Version 1

```
p1 = {};  
p1["x"] = 13  
p1["y"] = 7;  
console.assert(p1["x"] === 13);  
console.assert(p1["y"] === 7);
```

## Version 2

```
// Version 2: Syntactic sugar 1  
p2 = {"x": 13, "y": 7};  
console.assert(p1 !== p2);  
console.assert(p1["x"] === p2["x"]);  
console.assert(p1["y"] === p2["y"]);
```

# Syntactic sugar

## Version 1

```
p1 = {};  
p1["x"] = 13  
p1["y"] = 7;  
console.assert(p1["x"] === 13);  
console.assert(p1["y"] === 7);
```

## Version 3

```
p3 = new Object();  
p3.x = 13  
p3.y = 7;  
console.assert(p1.x === p3.x)  
console.assert(p1.y === p3.y);
```

## Version 2

```
// Version 2: Syntactic sugar 1  
p2 = {"x": 13, "y": 7};  
console.assert(p1 !== p2);  
console.assert(p1["x"] === p2["x"]);  
console.assert(p1["y"] === p2["y"]);
```

# Syntactic sugar

## Version 1

```
p1 = {};  
p1["x"] = 13  
p1["y"] = 7;  
console.assert(p1["x"] === 13);  
console.assert(p1["y"] === 7);
```

## Version 3

```
p3 = new Object();  
p3.x = 13  
p3.y = 7;  
console.assert(p1.x === p3.x)  
console.assert(p1.y === p3.y);
```

## Version 2

```
// Version 2: Syntactic sugar 1  
p2 = {"x": 13, "y": 7};  
console.assert(p1 !== p2);  
console.assert(p1["x"] === p2["x"]);  
console.assert(p1["y"] === p2["y"]);
```

## Summary

1. The usual dot-notation to get/set fields
2. `===` tests **references**; not structural
3. `new Object()` is a synonym for `{}`

# Expressions in $\lambda_{JS}$

# Expressions in $\lambda_{JS}$

$$e ::= \{\} \mid e[e] \mid e[e] \leftarrow e \mid e(e) \mid \lambda x.e$$

- **Objects are immutable!**
- **Only anonymous functions.**
- `{}` creates an "empty" object, that is, all fields are assigned to `undefined`
- The notation  $e_o[e_f]$  reads the field  $e_f$  from object  $e_o$
- The notation  $e_o[e_f] \leftarrow e_v$  is an immutable extension of a map. Recall `(hash-add ht x v)`.

# Runtime-only values



# Runtime-only values

$$v = (E, \lambda x.e) \mid O$$

- Closures are **run-time-only** values, so no way to show it in JavaScript
- Objects  $O$  are also **run-time-only** values and represents a map from strings to values (recall environments in  $\lambda_E$ )

# $\lambda_E$ -calculus

## Revisiting Lesson 12

$$v \Downarrow_E v \quad (\mathbf{E}\text{-val})$$

$$x \Downarrow_E E(x) \quad (\mathbf{E}\text{-var})$$

$$\lambda x.e \Downarrow_E (E, \lambda x.e) \quad (\mathbf{E}\text{-clos})$$

$$\frac{e_f \Downarrow_E (E_b, \lambda x.e_b) \quad e_a \Downarrow_E v_a \quad e_b \Downarrow_{E_b[x \mapsto v_a]} v_b}{(e_f \ e_a) \Downarrow_E v_b} \quad (\mathbf{E}\text{-app})$$

# Rules for objects

$$\frac{\forall s. O(s) = \text{undef}}{\{\} \Downarrow_E O} \text{E-empty}$$

$$\frac{e_o \Downarrow_E O \quad e_f \Downarrow_E s}{e_o[e_f] \Downarrow_E O(s)} \text{(E-get)}$$

$$\frac{e_o \Downarrow_E O \quad e_f \Downarrow_E s \quad e_v \Downarrow_E v}{e_o[e_f] \leftarrow e_v \Downarrow_E O[s \mapsto v]} \text{(E-set)}$$

# Implementation

# Syntax

$$c ::= n \mid s \mid b \mid \text{undef}$$

# Syntax

$$c ::= n \mid s \mid b \mid \text{undef}$$

```
(define (k:const? v)
  (or (k:number? v)
      (k:string? v)
      (k:bool? v)
      (k:undef? v)))
(struct k:number (value) #:transparent)
(struct k:string (value) #:transparent)
(struct k:bool (value) #:transparent)
(struct k:undef () #:transparent)
```

# Values

$$v = c \mid (E, \lambda x.e) \mid O$$

# Values

$$v = c \mid (E, \lambda x.e) \mid O$$

```
(define (j:value? v)
  (or (k:const? v)
      (j:closure? v)
      (j:object? v)))
(struct j:closure (env decl) #:transparent)
(struct j:object (data) #:transparent)
```



# Expressions

$$e ::= \lambda x.e \mid x \mid e(e) \mid e[e] \mid e[e] \leftarrow e$$

# Expressions

$$e ::= \lambda x.e \mid x \mid e(e) \mid e[e] \mid e[e] \leftarrow e$$

```
(struct j:lambda (params body) #:transparent)
(struct j:variable (name) #:transparent)
(struct j:apply (func args) #:transparent)
;; Object-related operations
(struct j:get (obj field) #:transparent)
(struct j:set (obj field value) #:transparent)
```

# Using the dot notation

■ In JavaScript we can use the dot notation.

```
var x = {} // Creates a new, empty object
// Reading an unset field returns undefined
console.assert(x.foo == undefined)
// Assignment works as expected
x.foo = 20
console.assert(x.foo == 20);
// Deleting a field is equivalent to assigning to undefined
delete x.foo; // x.foo = undefined ← in our semantics
console.assert(x.foo == undefined);
```

# Object creation

# Object creation

■ We can use functions to create objects.

```
function shape(x, y) {  
  return {"x": x, "y": y};  
}  
var p = shape(10, 2);  
console.assert(p.x === 10);  
console.assert(p.y === 2);
```

# Object creation

■ We can use functions to create objects.

```
function shape(x, y) {  
  return {"x": x, "y": y};  
}  
var p = shape(10, 2);  
console.assert(p.x == 10);  
console.assert(p.y == 2);
```

```
function rectangle(x, y, width, height) {  
  var obj = shape(x, y);  
  obj.width = width;  
  obj.height = height;  
  return obj;  
}  
var r = rectangle(0, 1, 10, 3);  
console.assert(r.x == 0);  
console.assert(r.y == 1);  
console.assert(r.width == 10);  
console.assert(r.height == 3);
```

# Revisiting object creation

Operator `new` can be combined with functions to create objects.

```
function Shape(x, y) {  
  this.x = x;  
  this.y = y;  
}  
p1 = new Shape(0, 1);  
console.assert(p1.x == 0);  
console.assert(p1.y == 1);
```

# Revisiting object creation

Operator `new` can be combined with functions to create objects.

```
function Shape(x, y) {  
  this.x = x;  
  this.y = y;  
}  
p1 = new Shape(0, 1);  
console.assert(p1.x == 0);  
console.assert(p1.y == 1);
```

```
function Shape(obj, x, y) {  
  obj.x = x;  
  obj.y = y;  
  return obj;  
}  
p1 = Shape({}, 0, 1);  
console.assert(p1.x == 0);  
console.assert(p1.y == 1);
```

We will revisit `new` and how to represent it in our interpreter.



# Object methods

We can use a function's closure to implement object method's (functions bound to a data-structure via `this`).

```
function Shape(x, y) {  
  this.x = x;  
  this.y = y;  
  this.translate = function(x, y) {  
    this.x += x;  
    this.y += y;  
  }  
}  
p1 = new Shape(0, 1);  
p1.translate(10, 20);  
console.assert(p1.x == 10);  
console.assert(p1.y == 21);
```

```
function Shape(obj, x, y) {  
  obj.x = x;  
  obj.y = y;  
  obj.translate = (x, y) => {  
    obj.x += x;  
    obj.y += y;  
  }  
  return obj;  
}  
p1 = Shape({}, 0, 1);  
p1.translate(10, 20);  
console.assert(p1.x == 10);  
console.assert(p1.y == 21);
```

# Method creation syntactic sugar

JavaScript includes some convenient syntax to declare classes, but semantically, this is just syntactic sugar.

```
class Shape {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
  translate(x, y) {  
    this.x += x;  
    this.y += y;  
  }  
}  
p1 = new Shape(0, 1);  
p1.translate(10, 20);  
console.assert(p1.x == 10);  
console.assert(p1.y == 21);
```

# Object Inheritance

# Class inheritance

JavaScript includes some convenient syntax to extend classes, but semantically, this feature is also syntactic sugar.

```
class Rectangle extends Shape {  
  constructor(width, height) {  
    super(0, 0);  
    this.width = width;  
    this.height = height;  
  }  
}  
  
var r1 = new Rectangle(10, 20);  
r1.translate(5, 6);  
console.assert(r1.x == 5);  
console.assert(r1.y == 6);
```

# Inheritance

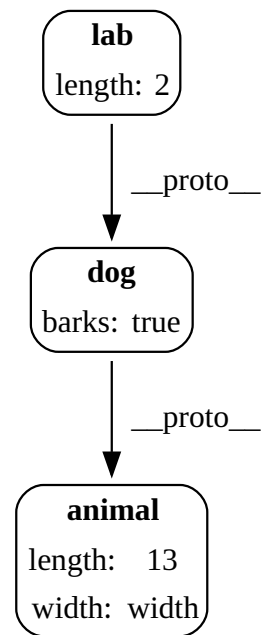
```
var animal = { "length": 13, "width": 7 }; // Source: Essence of JavaScript
console.assert(animal["length"] === 13);
console.assert(animal["width"] === 7);
console.assert(animal["foo"] === undefined);
```

// We can say that a dog is an animal, with the proto field

```
var dog = { "__proto__": animal, "barks": true };
console.assert(dog["barks"]);
console.assert(dog["length"] === 13);
console.assert(dog["width"] === 7);
console.assert(dog["foo"] === undefined);
```

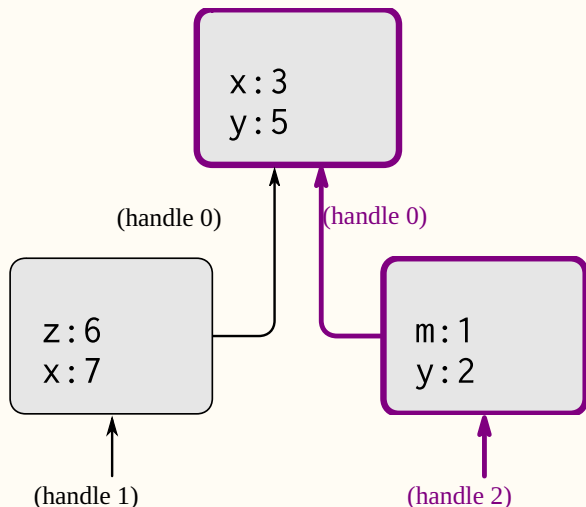
// We can then create a special kind of dog, a labrador

```
var lab = { "__proto__": dog, "length": 2 };
console.assert(lab["barks"]);
console.assert(lab["length"] === 2);
console.assert(lab["width"] === 7);
console.assert(lab["foo"] === undefined);
```



# Quiz

JavaScript objects can be thought of environments as first-class values.



List all variable bindings  
in object `h2`

```
let h0 = { "x": 3, "y": 5 };  
let h1 = { "z": 6, "x": 7, "__proto__": h0 }  
let h2 = { "m": 1, "y": 2, "__proto__": h0 }
```

**Figure 3.1:** A simple environment structure.

Source: SICP book Section 3.2

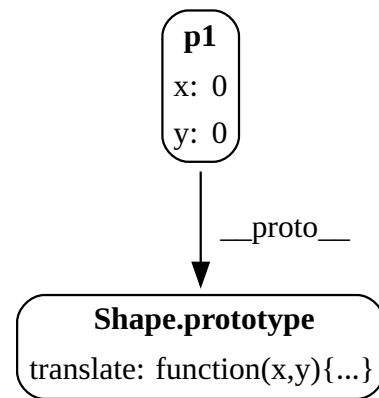
# JavaScript `__proto__` deprecated!

- Direct access to attribute `__proto__` is discouraged and deprecated!
- However, getting/setting attribute `__proto__` is syntactic sugar for `GetPrototypeOf` and `SetPrototypeOf` in the JavaScript specification.
- We are using `__proto__` mainly because we are following the Essence of JavaScript.
- Prototypes can be updated dynamically due to mutation

# JavaScript function objects

We can use field `prototype` to declare the prototype of a given class. We can also use field `prototype` to add methods to an object. Operation `new` assigns `Shape.prototype` to `p1.__proto__`.

```
function Shape(x, y) {  
  this.x = x;  
  this.y = y;  
}  
  
// This way we bind the method once  
Shape.prototype.translate = function (x, y) {  
  this.x += x;  
  this.y += y;  
}  
  
p1 = new Shape(0, 1);  
p1.translate(10, 20);  
console.assert(p1.x == 10);  
console.assert(p1.y == 21);
```





# Desugaring object inheritance

```
var Shape = (obj, x, y) => { // Shape's constructor
  obj.x = x;
  obj.y = y;
  return obj
}
Shape.prototype = {} // Shape extends Object
Shape.prototype.translate = function (x, y) { // Also add method translate
  this.x += x;
  this.y += y;
}
p1 = Shape({ "__proto__": Shape.prototype }, 0, 1); // When creating, init prototype
p1.translate(10, 20);
console.assert(p1.x == 10);
console.assert(p1.y == 21);
```

# Desugaring class creation

## Version 3

```
class Shape {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
  }  
  translate(x, y) {  
    this.x += x;  
    this.y += y;  
  }  
}  
p1 = new Shape(0, 1);
```

## Version 2

```
function Shape(x, y) {  
  this.x = x;  
  this.y = y;  
}  
Shape.prototype.translate =  
  function (x, y) {  
    this.x += x;  
    this.y += y;  
  }  
p1 = new Shape(0, 1);
```

## Version 1

```
Shape = (obj, x, y) => {  
  obj.x = x;  
  obj.y = y;  
  return obj  
}  
Shape.prototype = {}  
Shape.prototype.translate =  
  function (x, y) {  
    this.x += x;  
    this.y += y;  
  }  
p1 = Shape(  
  {"__proto__": Shape.prototype},  
  0, 1);
```

# Inheritance desugaring

```
class Rectangle extends Shape {  
  constructor(width, height) {  
    super(0, 0);  
    this.width = width;  
    this.height = height;  
  }  
}  
var r1 = new Rectangle(10, 20);
```

```
function Rectangle(width, height)  
  Shape.call(this, 0, 0);  
  this.width = width;  
  this.height = height;  
}  
Rectangle.prototype =  
  { "__proto__": Shape.prototype };  
var r1 = new Rectangle(10, 20);
```

```
Rectangle = (obj, w, h) => {  
  Shape(obj, 0, 0);  
  obj.width = w;  
  obj.height = h;  
  return obj;  
}  
Rectangle.prototype =  
  { "__proto__": Shape.prototype };  
r1 = Rectangle(  
  { "__proto__": Rectangle.prototype },  
  0, 1);
```

# Summary

- Introduced `--proto--`, which introduces prototype inheritance
- Introduced methods at the prototype level
- Introduced class extension
- Introduced syntactic desugaring