# CS450

## Structure of Higher Level Languages

### Lecture 37: Methods & object inheritance

Tiago Cogumbreiro

# Today we will...

- Implement JavaScript's inheritance mechanism
- Learn about prototype-based inheritance

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);
```

# 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

UMASS BOSTON

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.

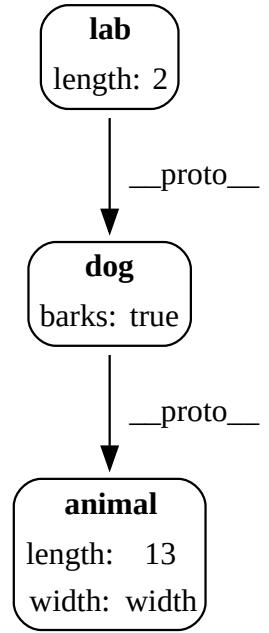
```
class Rectangle extends Shape {
  constructor(width, height) {
    super(0, 0);
    this.width = width;
    this.height = height;
  }
}
var r1 = new Rectangle(10, 20);
r1.translate(5,6);
console.assert(r1.x == 5);
console.assert(r1.y == 6);
```

# Inheritance

```
var animal = { "length": 13, "width": 7 }; // Source: Essence of JavaScrip
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.

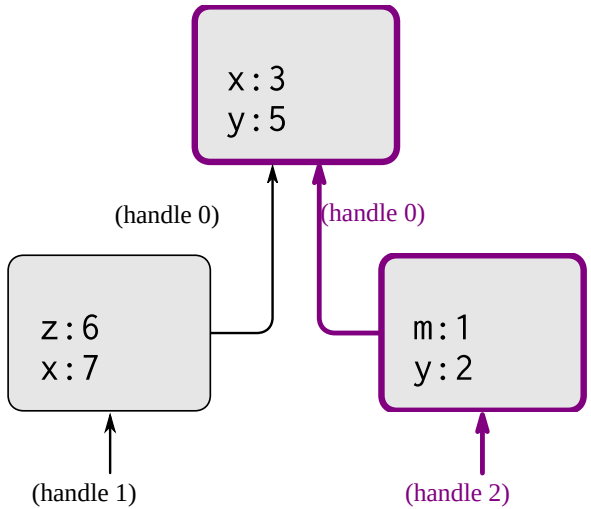



**Figure 3.1:** A simple environment structure.

Source: SICP book Section 3.2

## 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}
```

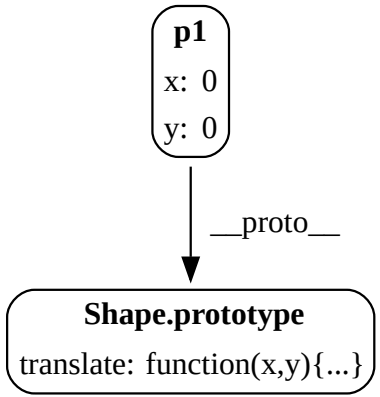
# JavaScript `__proto__` deprecated!

- 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