Objects (cont.)

Deian Stefan

(Adopted from my & Edward Yang's CS242 slides)

Today

- Continue with central OO concepts (JavaScript)
  - Objects
  - Dynamic dispatch/lookup
  - Encapsulation
  - Subtyping
  - Inheritance
  - Classes
- C++ vtables

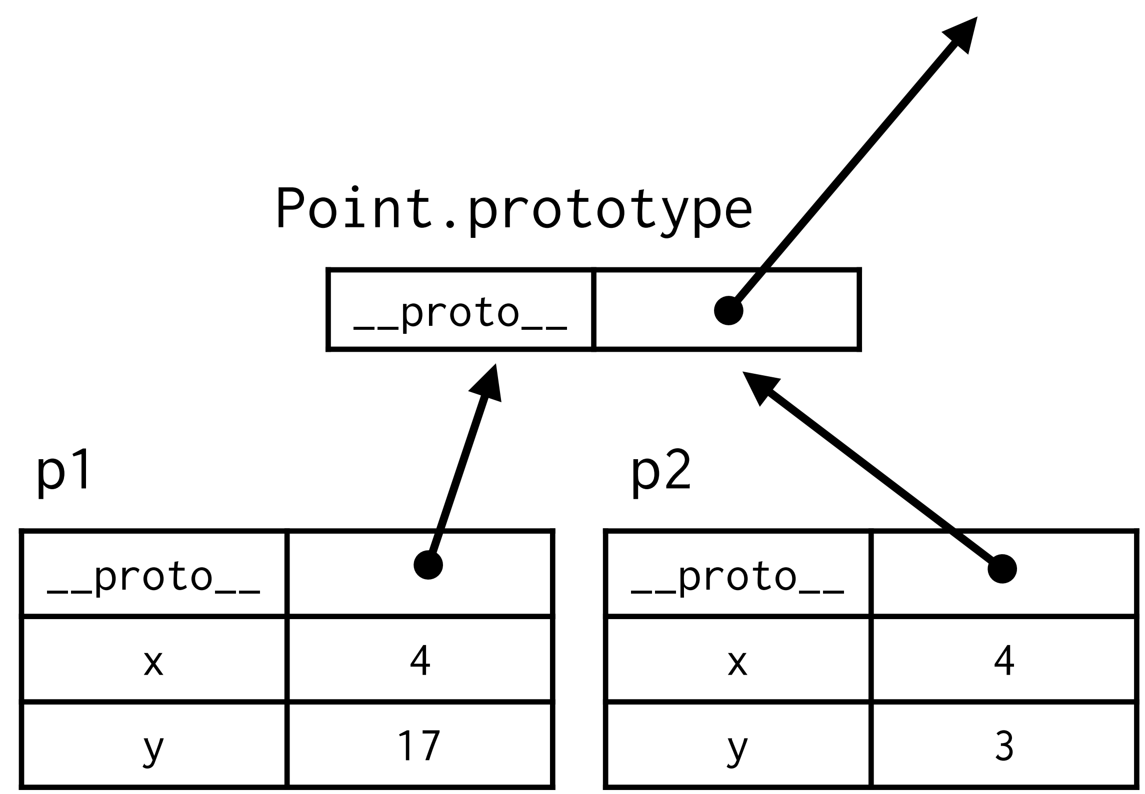
JavaScript objects

- Collection of properties (named values)
  - Data properties = "instance variables"
    - Retrieved by effectively sending **get message** to object
    - Assigned by effectively sending **set message** to object
  - Methods: properties where value is a JavaScript function
    - Can use `this` to refer to the object method was called on

Creating objects

- When invoking function with `new` keyword, runtime creates a new object and sets the receiver (`this`) to it before calling function

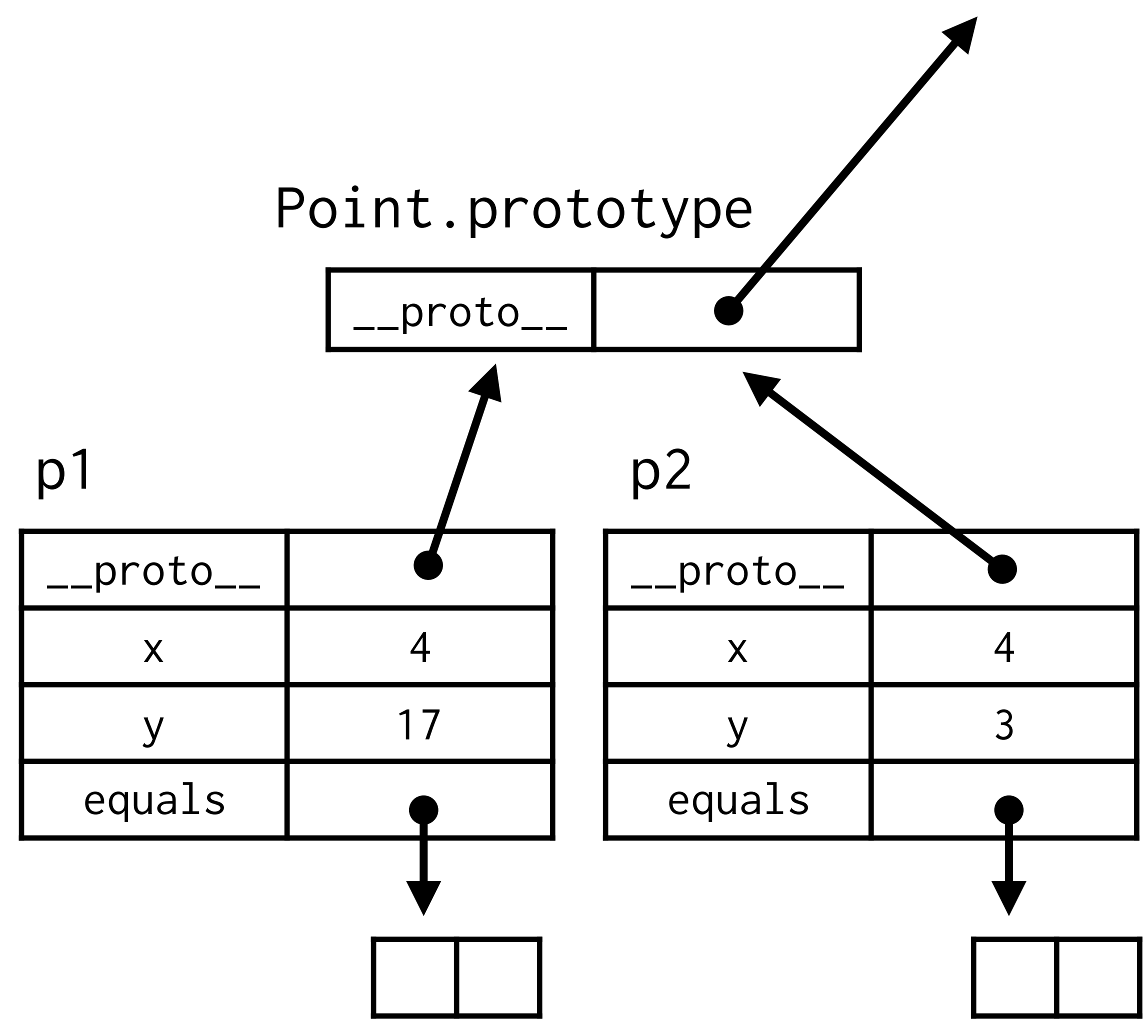
```
function Point(x, y) {
  this.x = x;
  this.y = y;
  return this;
 }

const p1 = new Point(4, 17);
const p2 = new Point(4, 3);
```



“Instance” Methods

- What if we want to compare objects? Do it this way?

```
function Point(x, y) {
  this.x = x;
  this.y = y;
  this.equals = function (p) {
    ...
  };
  return this;
 }

const p1 = new Point(4, 17);
const p2 = new Point(4, 3);
p1.equals(p2);
```

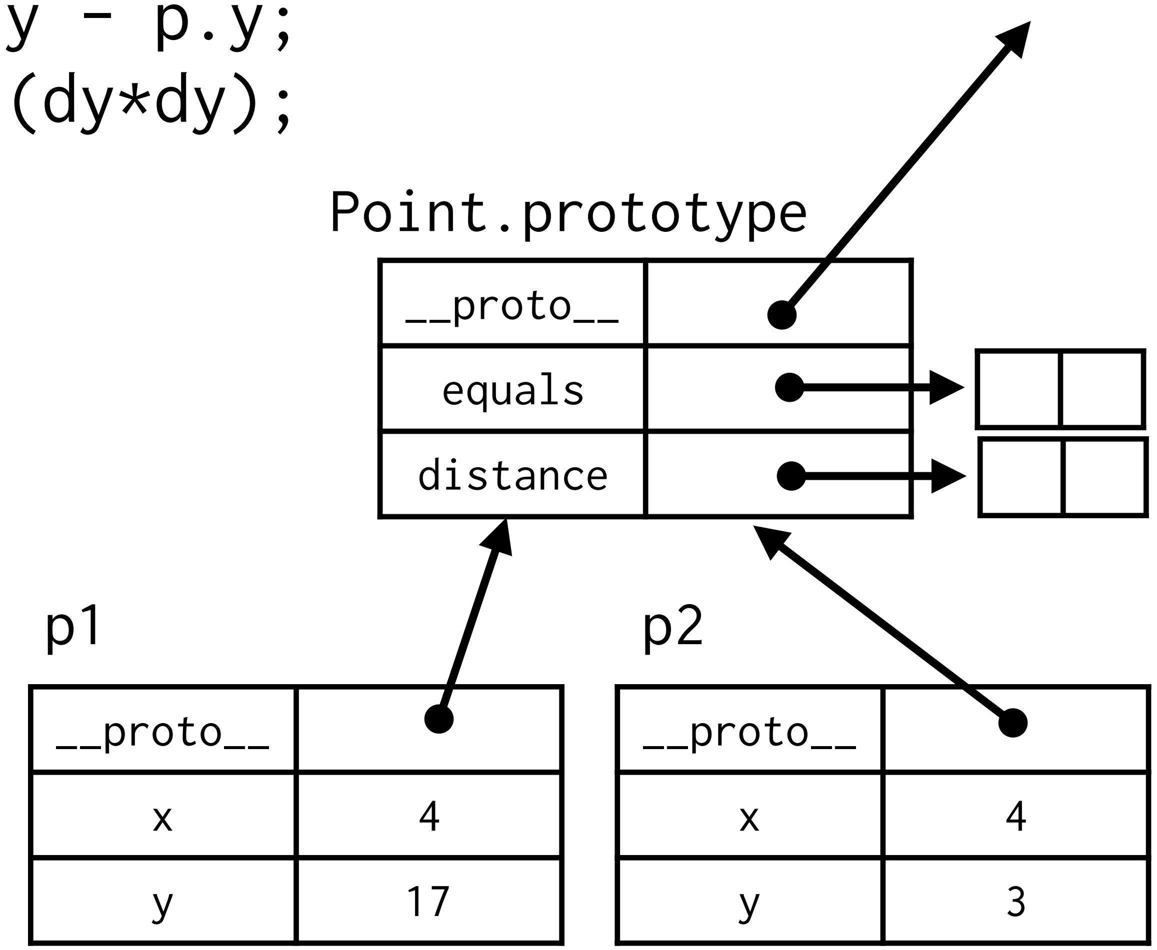


“Instance” Methods

- Every object has a prototype
  - Prototype is an object that serves as the “template” for objects of a common category
- When doing dynamic lookup:
  1. Check for property on object and if there return it
  2. Else, get the prototype of object and goto 1
- How do we get at prototype?
  - `Point.prototype` is the prototype of every `Point` object
  - You can get it via `Object.getPrototypeOf(pt)`

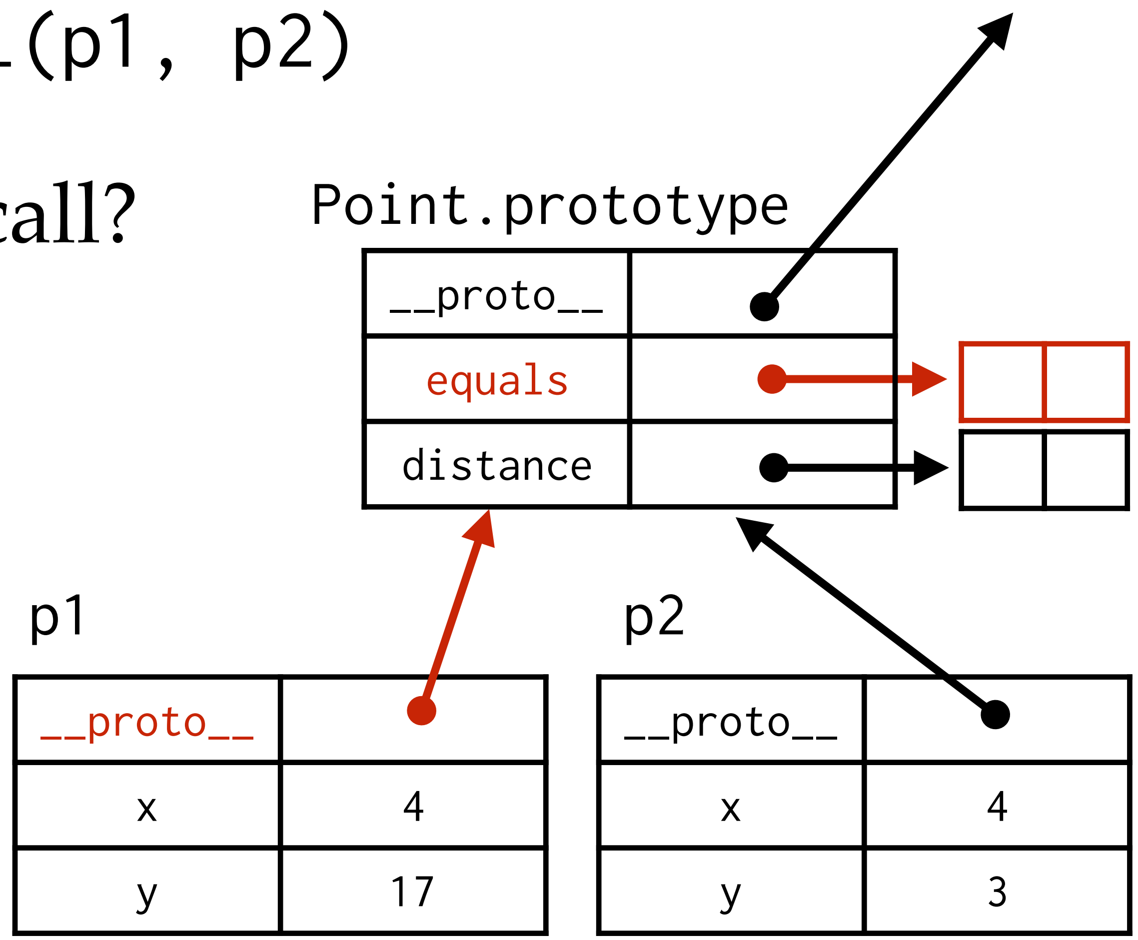
“Instance” Methods

```
Point.prototype.equals = function(p) {
  return Math.abs(this.x - p.x) +
         Math.abs(this.y - p.y) < 0.00001;
}

Point.prototype.distance = function(p) {
  const dx = this.x - p.x, dy = this.y - p.y;
  return Math.sqrt(dx*dx) + Math.sqrt(dy*dy);
}

const p1 = new Point(4, 17);
const p2 = new Point(4, 3);
p1.equals(p2);
```



Dynamic lookup

- Invoking method = sending message to object
  - Implementation: call function with receiver set to the object
  - E.g. `p1.equals(p2)` is equivalent to: `Point.prototype.equals.call(p1, p2)`
  - How do you find function to call?
- Dynamic lookup!
  - Chase prototypes until method is found



Dynamic lookup

Dynamic lookup

- What happens when a message is sent to an object and there is no corresponding method?
  - E.g., `p1.toHashValue();`

Dynamic lookup

- What happens when a message is sent to an object and there is no corresponding method?
  - E.g., `p1.toHashValue();`
- JavaScript has `Proxy` API that will let you intercept messages (get, set, delete, hasOwn, etc.)

How do proxies work?

- Define handlers and wrap object:

```
const handlers = {
        set: (target, property, value) => {
            ...
        },
        ...
};
let trappedObj = new Proxy(obj, handlers);
```

- How does this affect dynamic lookup?
- What is the cost of such a language feature?

Today

- Continue with central OO concepts (JavaScript)
  - Objects ✔
  - Dynamic dispatch/lookup ✔
  - Encapsulation
  - Subtyping
  - Inheritance
  - Classes
- C++ vtables

Encapsulation in JavaScript

- Methods are public
- Data is all public
- Can we do anything?

Subtyping in JavaScript

- What corresponds to an interface in JavaScript?
  - The properties of an object
- Subtyping in JavaScript is implicit; how so?
  - Can use any object as long as it has the expected properties

Subtyping in JavaScript

- What are cons and pros of implicit subtyping?
  - Pros: flexible in accepting any object that implements the right properties
  - Cons: relationship between objects not clear
- Subtyping imposes restrictions on dynamic dispatch; how so?
  - Must lookup properties based on names at runtime

Today

- Continue with central OO concepts (JavaScript)
  - Objects ✔
  - Dynamic dispatch/lookup ✔
  - Encapsulation ✔
  - Subtyping ✔
  - Inheritance
  - Classes
- C++ vtables

Inheritance in JavaScript

Let's make `ColoredPoint` inherit form `Point`:

```
ColoredPoint.prototype = Point.prototype;
```

- Is this correct? A: yes **B: no**
- Changing properties on ColoredPoint.prototype may clobber Point.prototype in unexpected ways

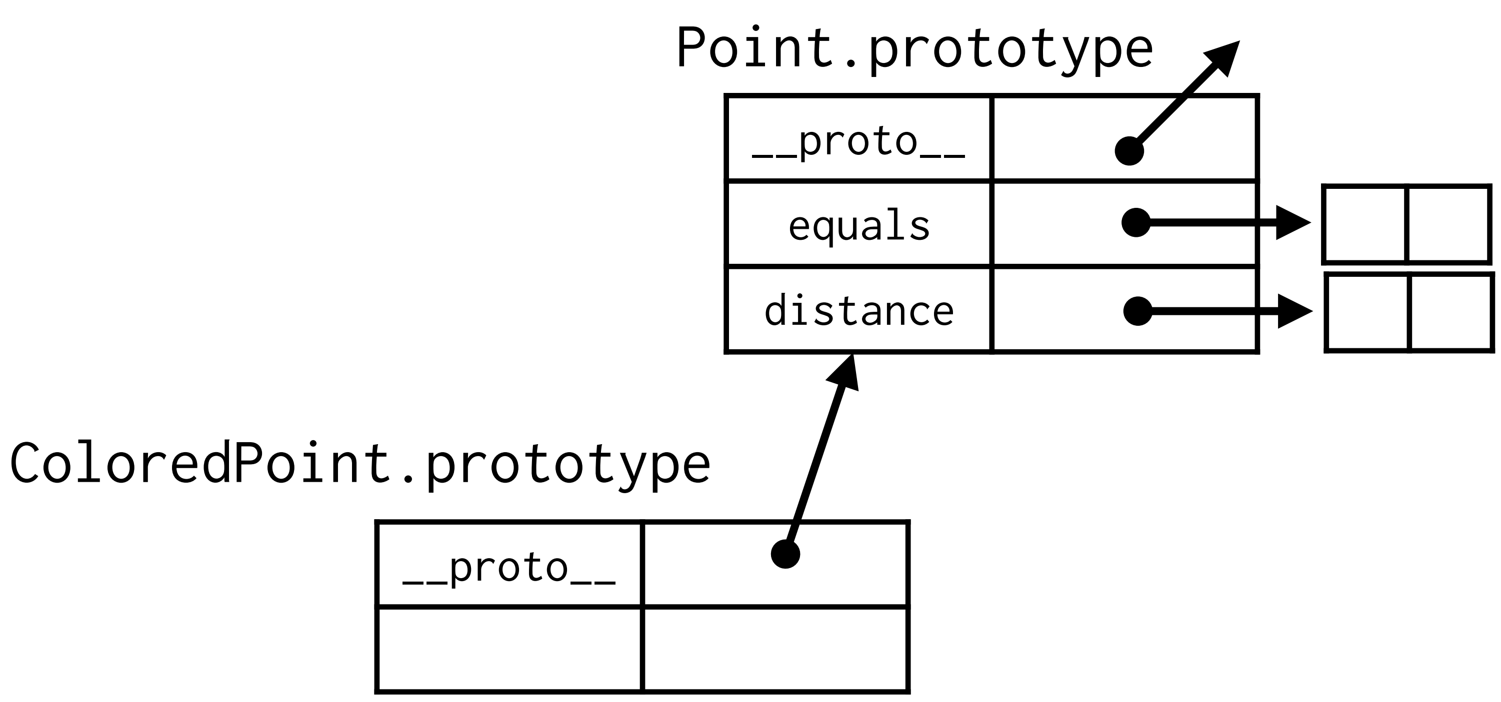
Inheritance in JavaScript

Let's make `ColoredPoint` inherit form `Point`:

- Correct approach:

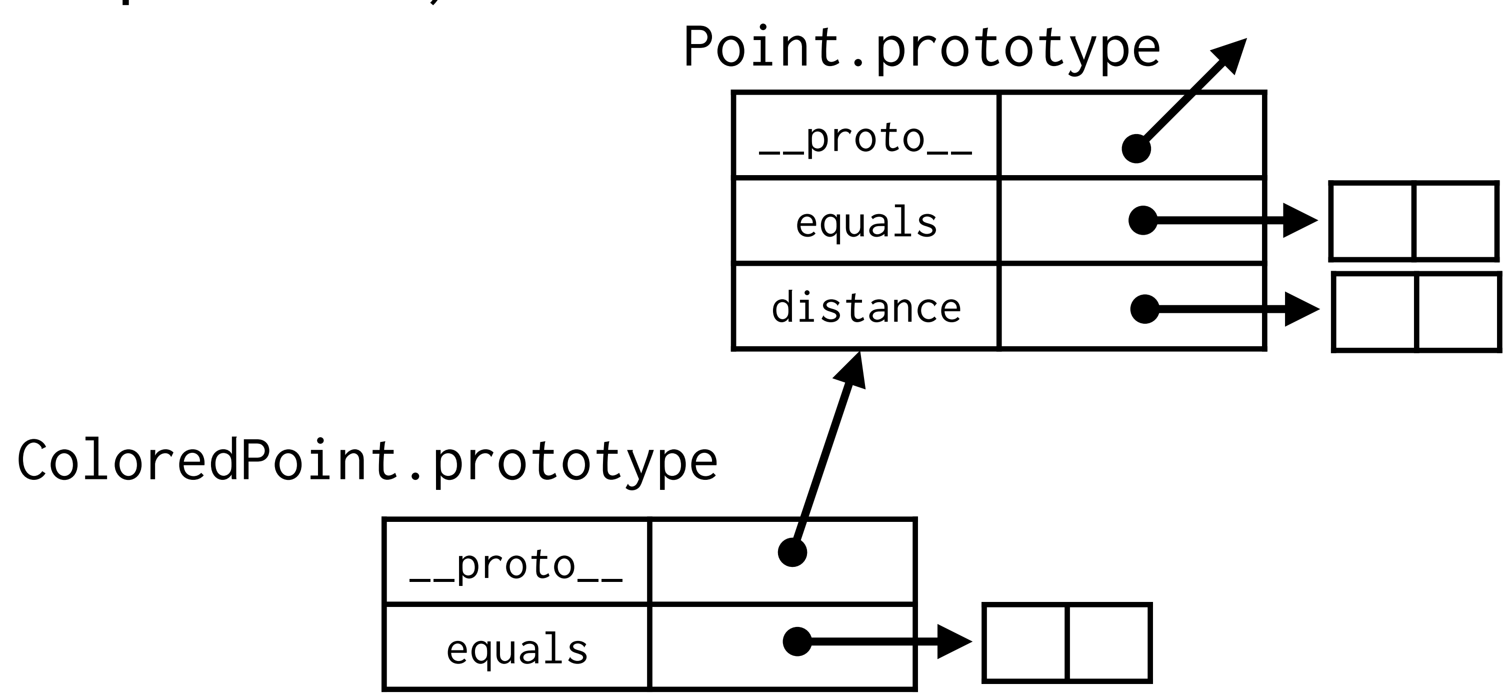
```
ColoredPoint.prototype = Object.create(Point.prototype);
```

- `Object.create` creates new object with specified prototype



Inheritance in JavaScript

```
function ColoredPoint(x, y, color) {
   Point.call(this, x, y);
   this.color = color;
}
ColoredPoint.prototype = Object.create(Point.prototype);
ColoredPoint.prototype.equals = function(p) {
  return (Math.abs(x - p.x) +
          Math.abs(y - p.y) < 0.00001)
          && color === p.color;
}
```



Inheritance in JavaScript

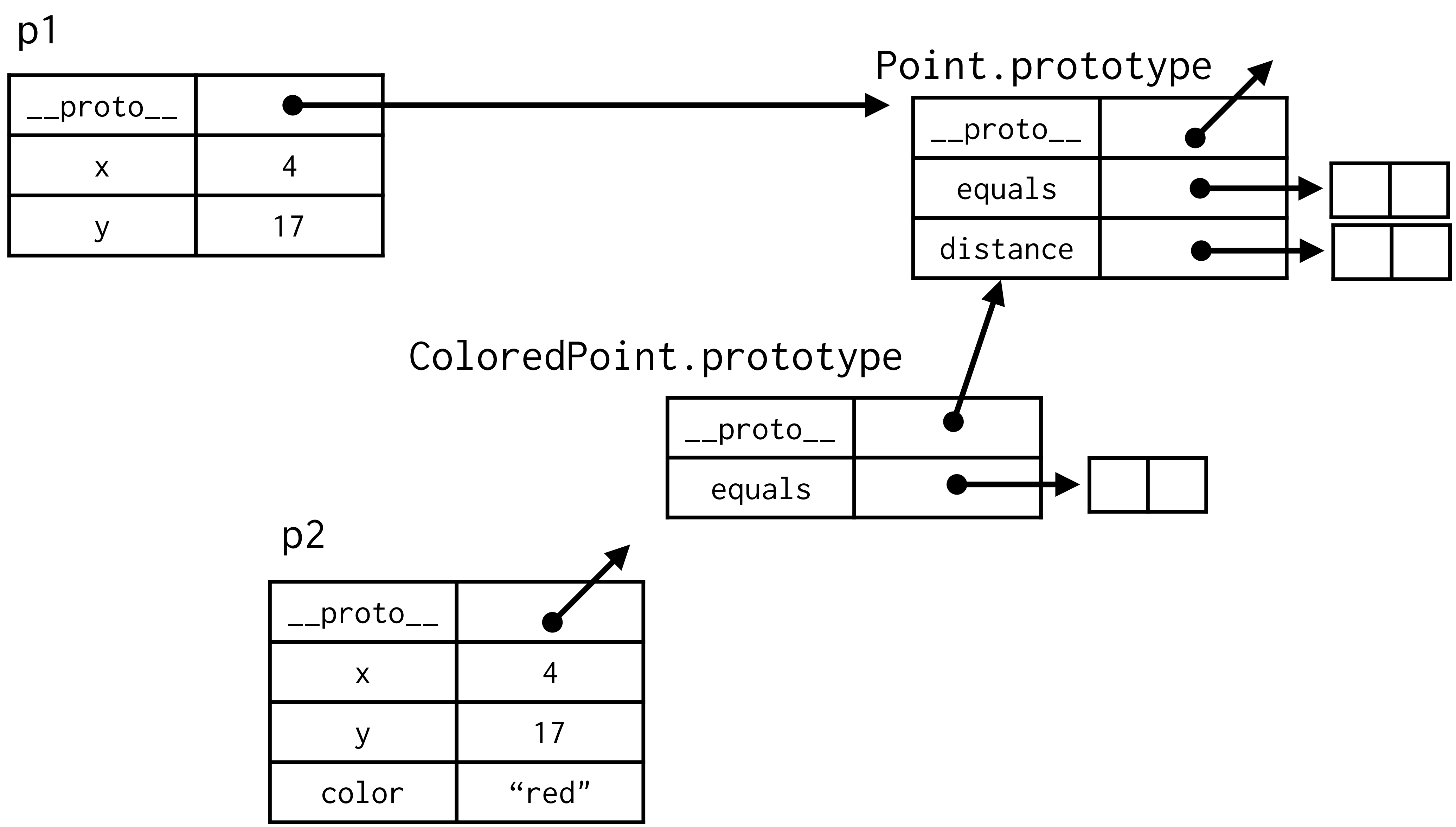
Could we have done it reverse order? A: yes, **B: no**

```
ColoredPoint.prototype.equals = function(p) {
  return (Math.abs(x - p.x) +
          Math.abs(y - p.y) < 0.00001)
          && color === p.color;
}
ColoredPoint.prototype = Object.create(Point.prototype);
```

This redefines the prototype to new object!

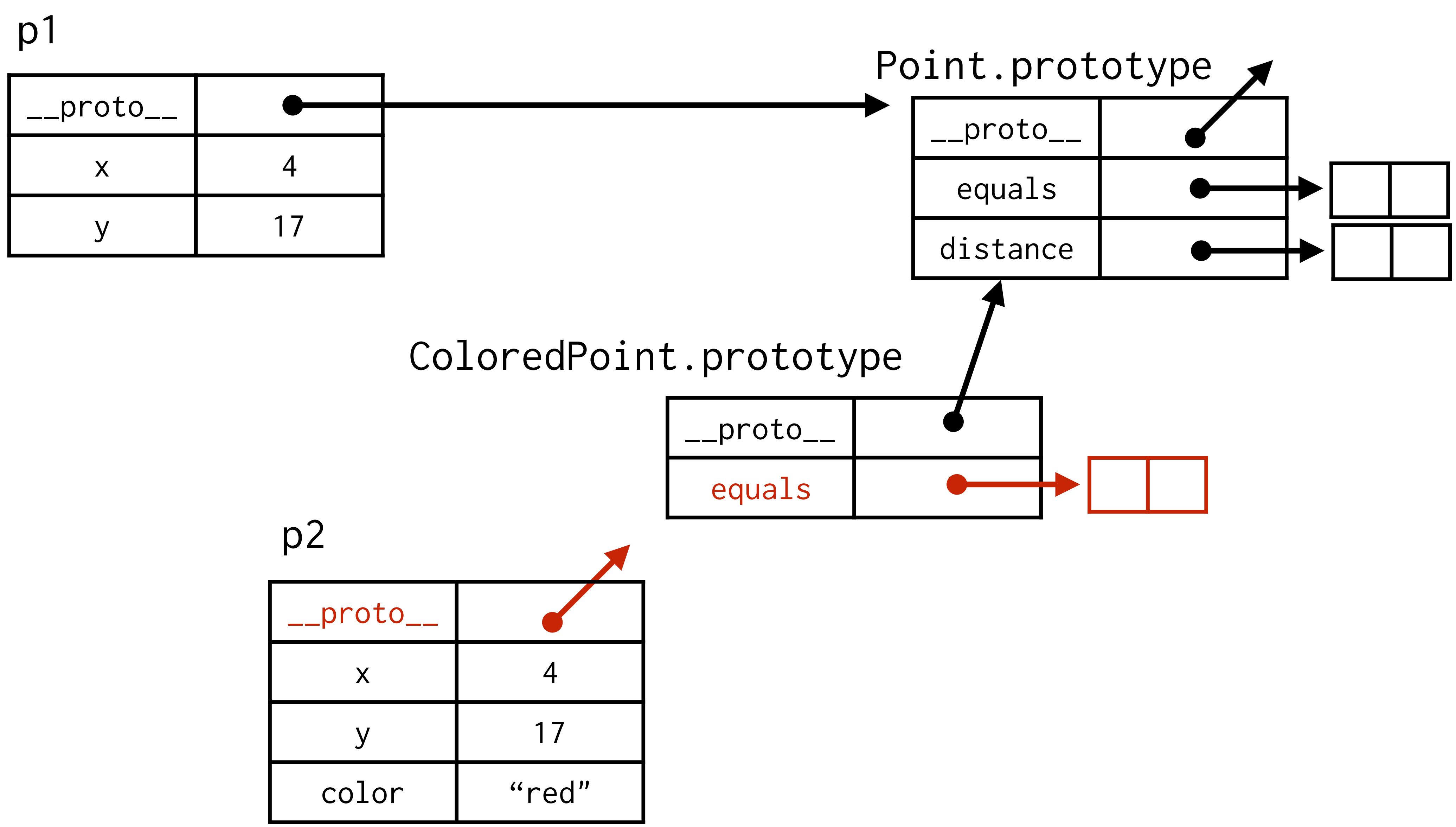
Inheritance in JavaScript

```
const p1 = new Point (4,17);
const p2 = new ColoredPoint (4,17,”red”);
p2.equals(p1);
p2.distance(p1);
```



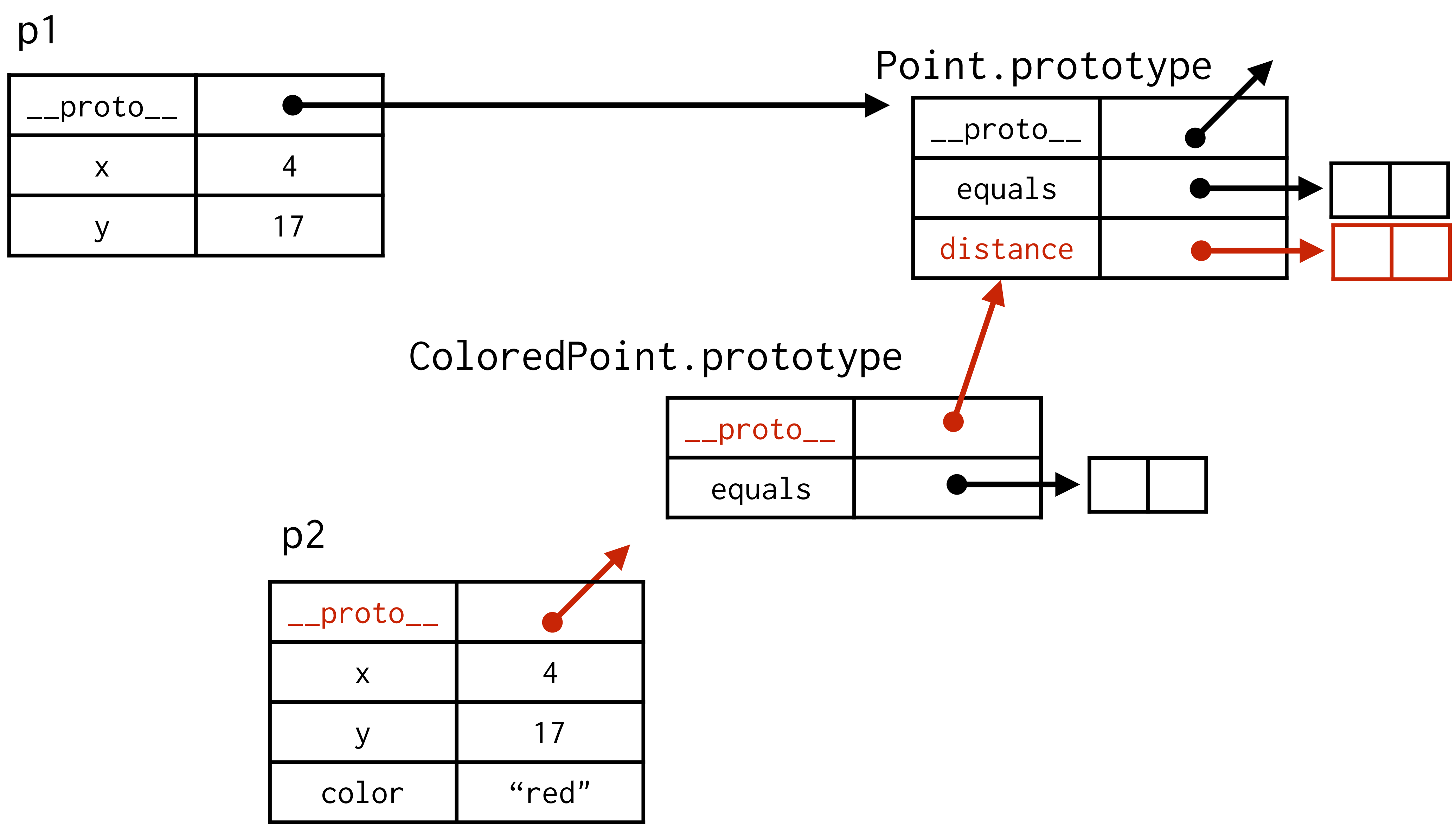
Inheritance in JavaScript

```
const p1 = new Point (4,17);
const p2 = new ColoredPoint (4,17,"red");
p2.equals(p1);
p2.distance(p1);
```



Inheritance in JavaScript

```
const p1 = new Point (4,17);
const p2 = new ColoredPoint (4,17,”red”);
p2.equals(p1);
p2.distance(p1);
```



Today

- Continue with central OO concepts (JavaScript)
  - Objects ✔
  - Dynamic dispatch/lookup ✔
  - Encapsulation ✔
  - Subtyping ✔
  - Inheritance ✔
  - Classes
- C++ vtables

What’s the deal with prototypes?

- Pros:
  - Open interfaces: can always extend object
  - Simple: single powerful mechanism
- Cons:
  - Slow: dynamic dispatch is name based
  - Not easy for to organize concepts OO style

JavaScript does have classes

- Why do we want language support for classes?
  - Make it easy/declarative to specify templates for objects
  - Don't add unnecessary dynamic checks (e.g., for new)
  - Make it easy to declare relationships (e.g., with new)
- In short, unified mechanism for:
  - Specification: name for describing contents
  - Implementation: template for creating new objects

Classes in JavaScript

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  equals(p) {
    return Math.abs(this.x - p.x) +
           Math.abs(this.y - p.y) < 0.00001;
  }
  distance(p) {
    const dx = this.x - p.x, dy = this.y - p.y;
    return Math.sqrt(dx*dx) + Math.sqrt(dy*dy);
  }
}
const p1 = new Point(4, 17);
const p2 = new Point(4, 3);
p1.equals(p2);
```

Classes in JavaScript

```
class ColoredPoint extends Point {
  constructor(x, y, color) {
   super(x, y);
   this.color = color;
  }
  equals(p) {
    return (Math.abs(x - p.x) +
        Math.abs(y - p.y) < 0.00001)
      && color === p.color;
  }
}
const p1 = new Point(4,17);
const p2 = new ColoredPoint(4,17,"red");
p1.equals(p2);
p1.distance(p2);
```

More on classes

- Classes are implemented using functions and proper setting of prototypes
- What are some benefits over vanilla functions?
  - Ensures constructor called with new
  - Provides support for inheritance by construction
  - Provides constructs like super and constructor to make things explicit and less error prone

JavaScript OO summary

- Objects: created by calling functions as constructors
- Encapsulation: no, must use closures or WeakMaps
- Dynamic dispatch: on object + prototype chasing
- Subtyping: implicit (based on handled messages)
- Inheritance: prototype hierarchy
- Classes: as of ES6, support for inheritance, super

Today

- Continue with central OO concepts (JavaScript)
  - Objects ✔
  - Dynamic dispatch/lookup ✔
  - Encapsulation ✔
  - Subtyping ✔
  - Inheritance ✔
  - Classes ✔
- C++ vtables

Why talk about C++?

- C++ is an OO extension of C
  - Borrows efficiency and flexibility
  - Borrows from Simula (OO program organization)
- Interesting design decisions
  - Features were and still are added incrementally
  - Backwards compatibility is a huge priority
  - “What you don’t use, you don’t pay for.”- Bjarne Stroustrup

Recall: C++ OO concepts in 1 slide

- Encapsulation
  - Public, private, protected + friend classes
- Dynamic lookup
  - Only for special functions: virtual functions
- Inheritance
  - Single and multiple inheritance!
  - Public and private base classes!
- Subtyping: tied to inheritance

Plan

- Look at dynamic lookup as done in C++ (vtables)
  - Why?
- Only interesting when inheritance comes into play
  - Why?

Simple example

```
class A {
  int a;
  void f(int);
}

A* pa;
pa->f(2);
```

Inheritance

```
class A {
  int a;
  void f(int);
}
class B {
  int b;
  void g(int)
}
class C {
  int c;
  void h(int)
}
```

Inheritance + virtual methods

```
class A {
  int a;
  virtual void f(int);
  virtual void g(int)
  virtual void h(int)
}
class B {
  int b;
  void g(int)
}
class C {
  int c;
  void h(int)
}
C* pc;
pc->g(2);
```

Difference from JavaScript

- Smalltalk and JavaScript: no static type system
  - In message `obj.method(arg)`, the `obj` can refer to anything
  - Need to find method using pointer from `obj`
  - The location in dictionary/hashtable will vary
- In C++ compiler knows the superclass for `obj`
  - Offset of data and function pointers are the same in subclass and superclass
  - Invoke function pointer at fixed offset in vtable!

Today

- Continue with central OO concepts (JavaScript)
  - Objects ✔
  - Dynamic dispatch/lookup ✔
  - Encapsulation ✔
  - Subtyping ✔
  - Inheritance ✔
  - Classes ✔
- C++ vtables ✔

Bonus

Virtual methods can be redefined

```
class A {
  int a;
  virtual void f() {
    printf(“parent”);
  }
}
class B {
  int b;
  virtual void f() {
    printf(“child”);
  }
}
A* pa = new B();
pa->f();
```

Non-virtual functions cannot

```
class A {
  int a;
  void f() {
    printf("parent");
  }
}
class B {
  int b;
  void f() {
    printf("child");
  }
}
A* pa = new B();
pa->f();
```