

Objects

Deian Stefan

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



Outline

- Central concepts in OO languages
- Objects as activation records (Simula)
- Dynamically-typed object-oriented languages
 - Class-based languages (Smalltalk)
 - Prototype-based languages (JavaScript)

Central concepts in OO languages

1. Dynamic lookup
2. Encapsulation
3. Subtyping
4. Inheritance

What are examples of objects?

What are examples of objects?

- File system

```
#include <unistd.h>
```

```
int open(const char *path, int oflag, ...);  
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

- DOM Elements

```
var log = document.getElementById("log");  
log.textContent = "w00t w00t";
```

- Integer

3 + 44

etc.

What is an object?

send a message
(method invocation) →

| hidden data | |
|------------------|---------------------|
| msg ₁ | method ₁ |
| ... | ... |
| msg ₂ | method ₂ |

- How is this different from ADTs?

What is an object?

send a message
(method invocation) →

| hidden data | |
|------------------|---------------------|
| msg ₁ | method ₁ |
| ... | ... |
| msg ₂ | method ₂ |

- How is this different from ADTs?
 - Behavioral not structural

Terminology

- **Selector:** name of a message (method name)
 - E.g., remove
- **Message:** selector + arguments
 - E.g., remove("log")
- **Method:** code used when responding to message
 - E.g.,

```
Array.prototype.remove = function (val) {  
    var i;  
    while((i == this.indexOf(val)) !== -1)  
        this.splice(i,1);  
    return this;  
}
```


1. Dynamic lookup

`object.message(args)`

- Invoke operation on object
 - Smalltalk: send message to object
 - C++: call member function on object
- Method is selected dynamically
 - Run-time operation
 - Depends on implementation of the object receiving the message

Is dynamic lookup = overloading?

- A: yes
- B: no

Dynamic lookup \neq overloading

- In overloading we can use the same symbol to refer to different implementations

- E.g., $1 + 1$ and $1.0 + 1.0$ use different implementations:

```
instance Num Int where  
  (+) = intPlus  
  ...
```

```
instance Num Float where  
  (+) = floatPlus  
  ...
```

- How is dynamic look different from this?

Dynamic lookup \neq overloading

- Consider:

```
for(var i = 0; i < arrA.length; i++) {  
    ... arrA[i] + arrB[i] ...  
}
```

- Here: send message `+arrB[i]` to object `arrA[i]`
- Which `+` we use is determined at run-time! Why?

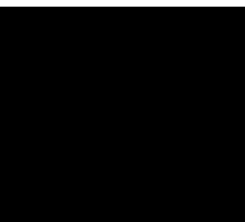
Dynamic lookup \neq overloading

- Overloading
 - Meaning of `operation(args)` is always the same
 - Code to be executed is resolved at compile-time
- Dynamic lookup
 - Meaning of `object.message(args)` depends on both object and message
 - Code to be executed is resolved at run-time

2. Abstraction / Encapsulation

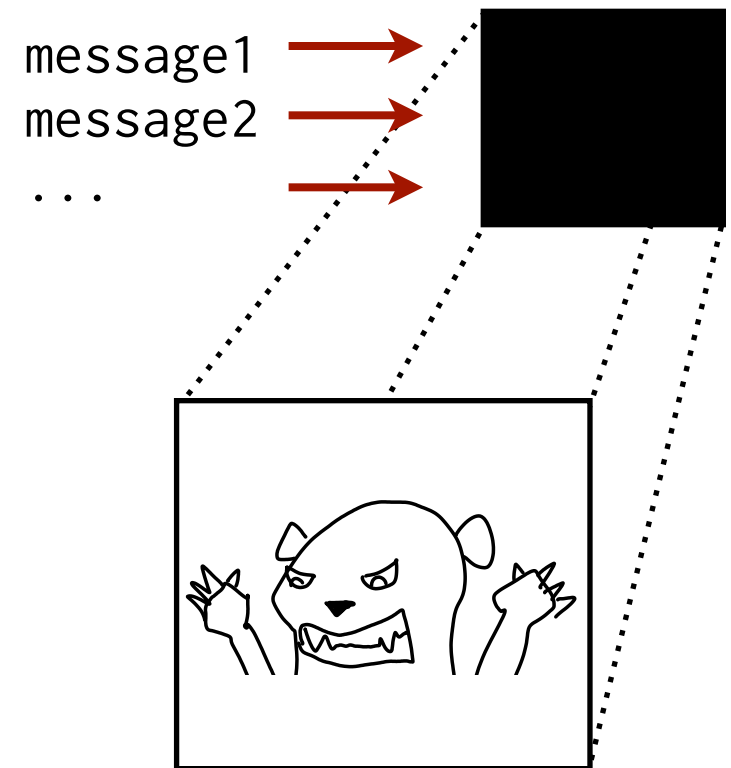
- Restricting access to a program component according to its specified interface
- Encapsulation separates views of
 - User of a component (has “abstract” view)
 - Operates by applying fixed set of operations provided by builder of abstraction
 - Builder of component (has detailed view)
 - Operates on representation

message1 →
message2 →
... →



2. Abstraction / Encapsulation

- Restricting access to a program component according to its specified interface
- Encapsulation separates views of
 - User of a component (has “abstract” view)
 - Operates by applying fixed set of operations provided by builder of abstraction
 - Builder of component (has detailed view)
 - Operates on representation



3. Subtyping

- Interface: external view of object
 - Messages understood by object (i.e., its type)
 - E.g.,
`interface(Point) == ["x", "y", "move"]`
`interface(ColorPoint) == ["x", "y", "move", "color"]`
- Subtyping is a relation ($<:$) between interfaces
 - If interface of A objects contains the whole interface of B object: A objects can be used where B objects are expected
 - We say A is a subtype of a B: $A <: B$
 - E.g., `ColoredPoint <: Point`

4. Inheritance



Same as subtyping?

- A: yes
- B: no

4. Inheritance

- Implementation: internal representation of object
 - Code for methods and supporting mechanism
- Inheritance: language feature that allows code reuse
 - New objects may be defined by reusing implementation of other objects
 - E.g., ColoredPoint implementation of move can reuse code used to implement move for Point objects

Subtyping implies inheritance?

- A: yes
- B: no

Subtyping \neq inheritance

Point:

x

y

move

ColoredPoint:

x

y

move

color

Subtyping \neq inheritance

Point:

x

y

move

\Rightarrow

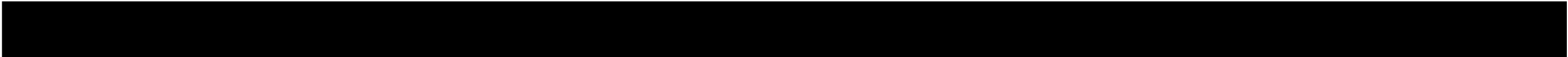
ColoredPoint:

x

y

move

color



Subtyping \neq inheritance

Point:

x

y

move

\Rightarrow

ColoredPoint:

x

y

move

color

```
Point.prototype.move =  
  function(dx, dy) {  
    this.x += dx;  
    this.y += dy;  
  }
```

Subtyping \neq inheritance

Point:

x

y

move

$\therefore >$

ColoredPoint:

x

y

move

color

```
Point.prototype.move =  
  function(dx, dy) {  
    this.x += dx;  
    this.y += dy;  
  }
```

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


Subtyping \neq inheritance

Point:

x

y

move

ColoredPoint:

x

y

move

color

\Rightarrow

```
Point.prototype.move =  
function(dx, dy) {  
  this.x += dx;  
  this.y += dy;  
}
```

```
ColoredPoint.prototype.move =  
function(dx, dy) {  
  this.x += dx+Math.random();  
  this.y += dy+Math.random();  
}
```

Subtyping \neq inheritance

Point:

x
y
move

ColoredPoint:

x
y
move
color

$\therefore >$

NO INHERITANCE!

```
Point.prototype.move =  
function(dx, dy) {  
  this.x += dx;  
  this.y += dy;  
}
```

```
ColoredPoint.prototype.move =  
function(dx, dy) {  
  this.x += dx+Math.random();  
  this.y += dy+Math.random();  
}
```

Inheritance implies subtyping?

- A: yes
- B: no

Why do we care about these?

- Dynamic lookup
 - In function-oriented programs, functions that operate on different kinds of data: need to select correct operations
- Abstraction, subtyping, inheritance
 - Organize system according to component interfaces
 - Extend system concepts / components
 - Reuse implementation through inheritance

Outline

- Central concepts in object-oriented languages
- ➔ Objects as activation records (Simula)
- Dynamically-typed object-oriented languages
 - Class-based languages (Smalltalk)
 - Prototype-based languages (JavaScript)

Objects as activation records

- Idea: after a function call is executed, leave the activation record on the stack, return pointer to it
 - E.g., Constructing objects in a JavaScript-like language

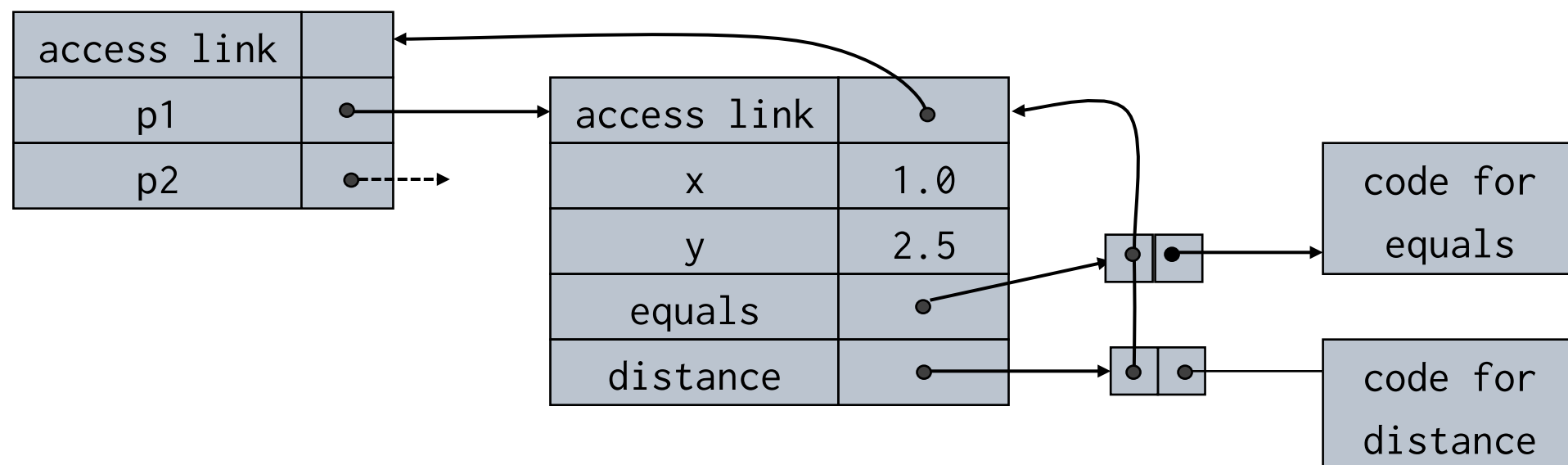
```
class Point(x, y) {  
  let equals = function (p) {  
    return Math.abs(x - p.x) +  
      Math.abs(y - p.y) < 0.00001;  
  }  
  let distance = function (p) {  
    var dx = x - p.x, dy = y - p.y;  
    return Math.sqrt(dx*dx) + Math.sqrt(dy*dy);  
  }  
}
```

Objects as activation records

- Add syntax for calling class & accessing object methods

```
let p1 = new Point (1.0, 2.5);  
let p2 = new Point (2.0, 2.5);  
p1.equals(p2);
```

- After executing first line:



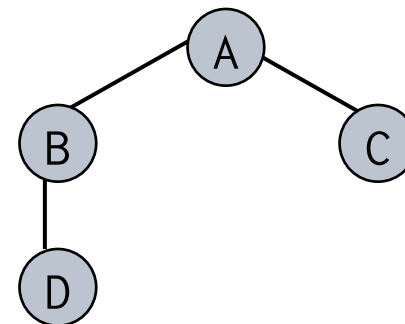
Simula

- First object-oriented language
 - Inspired many later designs, including Smalltalk and C++
- Objects in Simula
 - **Class:** function returning a pointer to its activation record
 - **Object:** instance of class, i.e., activation record produced by call to class
 - **Object access:** access any local variable / function using dot-notation: `object.var`
 - **Memory management:** garbage collect activation records

Derived classes in Simula

- A class declaration can be prefixed by a class name

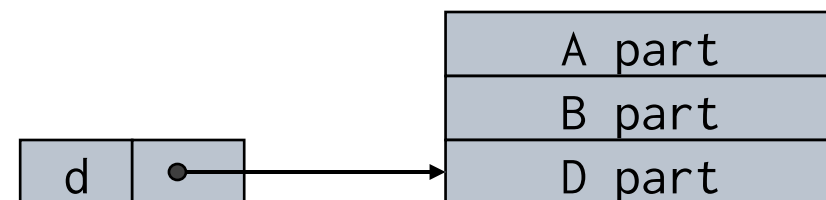
➤ E.g., class A
A class B
A class C
B class D



- An object of a “prefixed class” is the concatenation of objects of each class in prefix

➤ Inheritance & subtyping

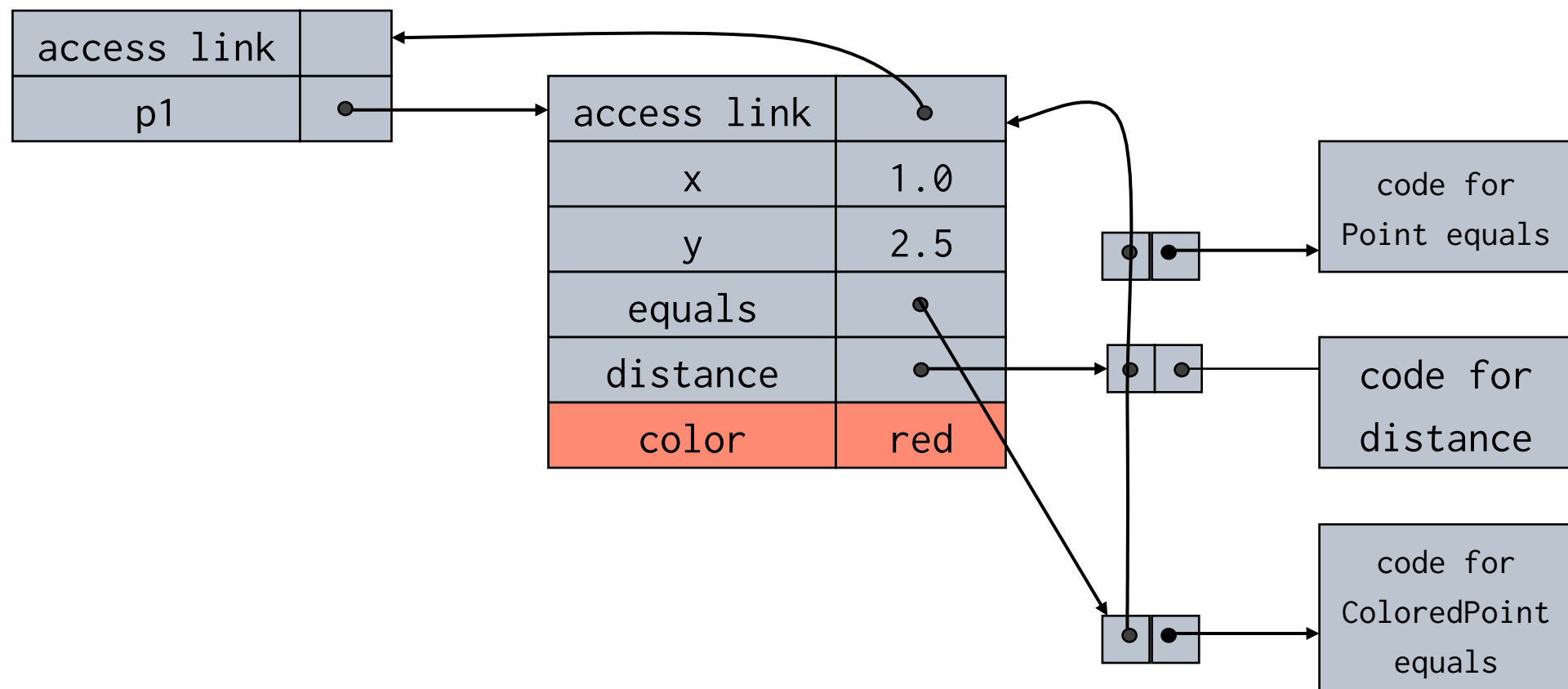
➤ E.g., `d = new D(...)`



- We say D is a *subclass* of B and B is a *superclass* of D

Prefix classes

```
Point class ColoredPoint(color) {  
  let equals = function (p) {  
    return (Math.abs(x - p.x) +  
            Math.abs(y - p.y) < 0.00001)  
            && color == p.color;  
  }  
}  
var p1 = new ColoredPoint(1.0, 2.5, "red");
```



Simula summary

- Main OO features
 - Classes: function that returns pointer to its activation record
 - Objects: activation record produced by call to class
 - Subtyping & inheritance: class hierarchy & prefixing
- Missing features
 - Encapsulation: all data and functions accessible
 - No notion of self / super (discussed in next few slides)

Outline

- Central concepts in object-oriented languages
- Objects as activation records (Simula)

Dynamically-typed object-oriented languages

- Class-based languages (Smalltalk)
- Prototype-based languages (JavaScript)

Smalltalk

- Object-oriented language
 - Everything is an object, even classes
 - All operations are messages to objects
 - Popularized objects
- The weird parts
 - Intended for “non-programmer”
 - Syntax presented by language-specific editor

FIGURE 11.19 GRAIL

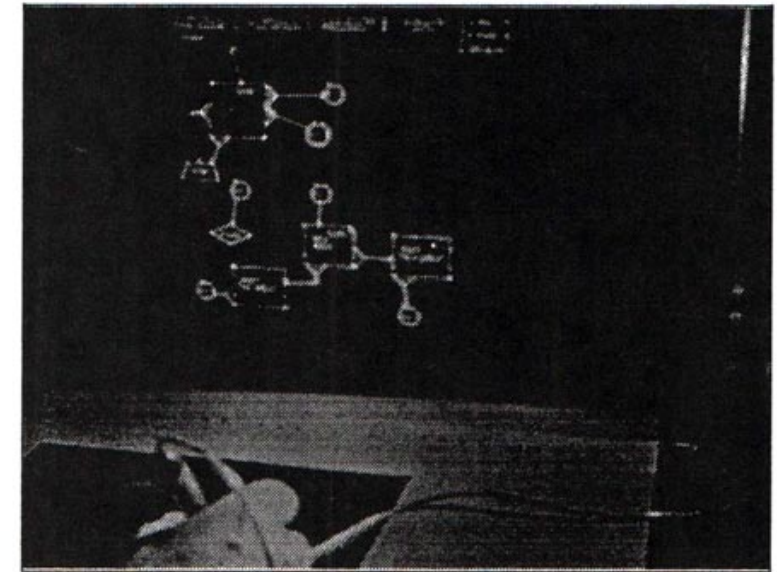


FIGURE 11.20 Seymour Papert and LOGO Turtle

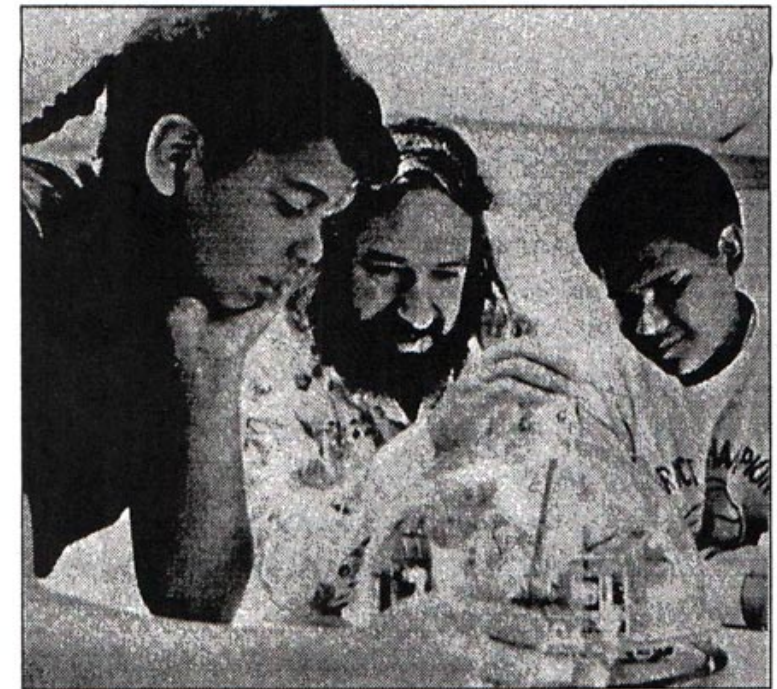


FIGURE 11.21 The Dynabook model



Smalltalk terminology

- **Class:** Defines behavior of its objects
- **Object:** Instance of some class
- **Selector:** name of a message
- **Message:** selector + arguments
- **Method:** code used when responding to message
- **Instance variable:** Data stored in object
- **Subclass:** Class defined by giving incremental modifications to some superclass

Smalltalk semantics

- Everything is an object
- Object communicate by sending / receiving messages
- Objects have their own state
- Every object is an instance of a class
- A class provides behavior for its instances

Example: Points

- Written in language-specific editor, in tabular form:

| | |
|-------------------------------|--------|
| class name | Point |
| super class | Object |
| class variables | pi |
| instance variables | x y |
| class messages and methods | |
| ⟨message & methods⟩ | |
| instance messages and methods | |
| ⟨message & methods⟩ | |

Example: Points

- Written in language-specific editor, in tabular form:

| | |
|-------------------------------|--------|
| class name | Point |
| super class | Object |
| class variables | pi |
| instance variables | x y |
| class messages and methods | |
| ⟨message & methods⟩ | |
| instance messages and methods | |
| ⟨message & methods⟩ | |

Instance messages and methods

- Getters and setters
 - Smalltalk does not have public variables
 - Get x-coordinate: `pt x`
 - Set new coordinates: `pt x:5 y:3`
- “Normal” methods
 - Move point: `pt moveDx:4 Dy: 5`
 - Draw point: `pt draw`

Instance messages and methods

- Getters and setters

- Smalltalk does not have public variables
- Get x-coordinate: pt `x`
- Set new coordinates: pt `x:5 y:3`

- “Normal” methods

mixfix selectors

- Move point: pt `moveDx:4 Dy: 5`
- Draw point: pt `draw`

Instance messages and methods

```
x || ^ x
y || ^ y
x:xcoord y:ycoord ||
  x <- xcoord
  y <- ycoord
moveDx:dx Dy:dy ||
  x <- x + dx
  y <- y + dy:dy
draw ||
  ...
```

Instance messages and methods

```
x || ^ x
y || ^ y
x:xcoord y:ycoord ||
  x <- xcoord
  y <- ycoord
moveDx:dx Dy:dy ||
  x <- x + dx
  y <- y + dy:dy
draw ||
...
```

New local scope

Instance messages and methods

```
x || ^ x
y || ^ y
x:xcoord y:ycoord ||
  x <- xcoord
  y <- ycoord
moveDx:dx Dy:dy ||
  x <- x + dx
  y <- y + dy:dy
draw ||
  ...
```



Instance variables

Instance messages and methods

```
x || ^ x
y || ^ y
x:xcoord y:ycoord ||
  x <- xcoord
  y <- ycoord
moveDx:dx Dy:dy ||
  x <- x + dx
  y <- y + dy:dy
draw ||
  ...
```

Mutable assignment

Instance messages and methods

```
x ||  x
y ||  y
x:xcoord y:ycoord ||
  x <- xcoord
  y <- ycoord
moveDx:dx Dy:dy ||
  x <- x + dx
  y <- y + dy:dy
draw ||
  ...
```

Return

Example: Points

| | |
|-------------------------------|--------|
| class name | Point |
| super class | Object |
| class variables | pi |
| instance variables | x y |
| class messages and methods | |
| ⟨message & methods⟩ | |
| instance messages and methods | |
| ⟨message & methods⟩ | |

Example: Points

| | |
|-------------------------------|--------|
| class name | Point |
| super class | Object |
| class variables | pi |
| instance variables | x y |
| class messages and methods | |
| ⟨message & methods⟩ | |
| instance messages and methods | |
| ⟨message & methods⟩ | |

Class messages and methods

- Class are objects too!
 - self is overloaded: always points to actual object

```
newOrigin ||  
  ^ self new x: 0 y: 0
```

```
newX:xvalue Y:yvalue ||  
  ^ self new x: xvalue y: yvalue
```

```
initialize ||  
  pi <- 3.14159
```

Class messages and methods

- Class are objects too!
 - self is overloaded: always points to actual object

```
newOrigin ||  
  ^ self new x: 0 y: 0
```

new message on self (Point class)

```
newX:xvalue Y:yvalue ||  
  ^ self new x: xvalue y: yvalue
```

```
initialize ||  
  pi <- 3.14159
```

Class messages and methods

- Class are objects too!
 - self is overloaded: always points to actual object

```
newOrigin ||  
^ self new x: 0 y: 0
```

new message on self (Point class)

x:0 y: 0 message on new Point obj

```
newX:xvalue Y:yvalue ||  
^ self new x: xvalue y: yvalue
```

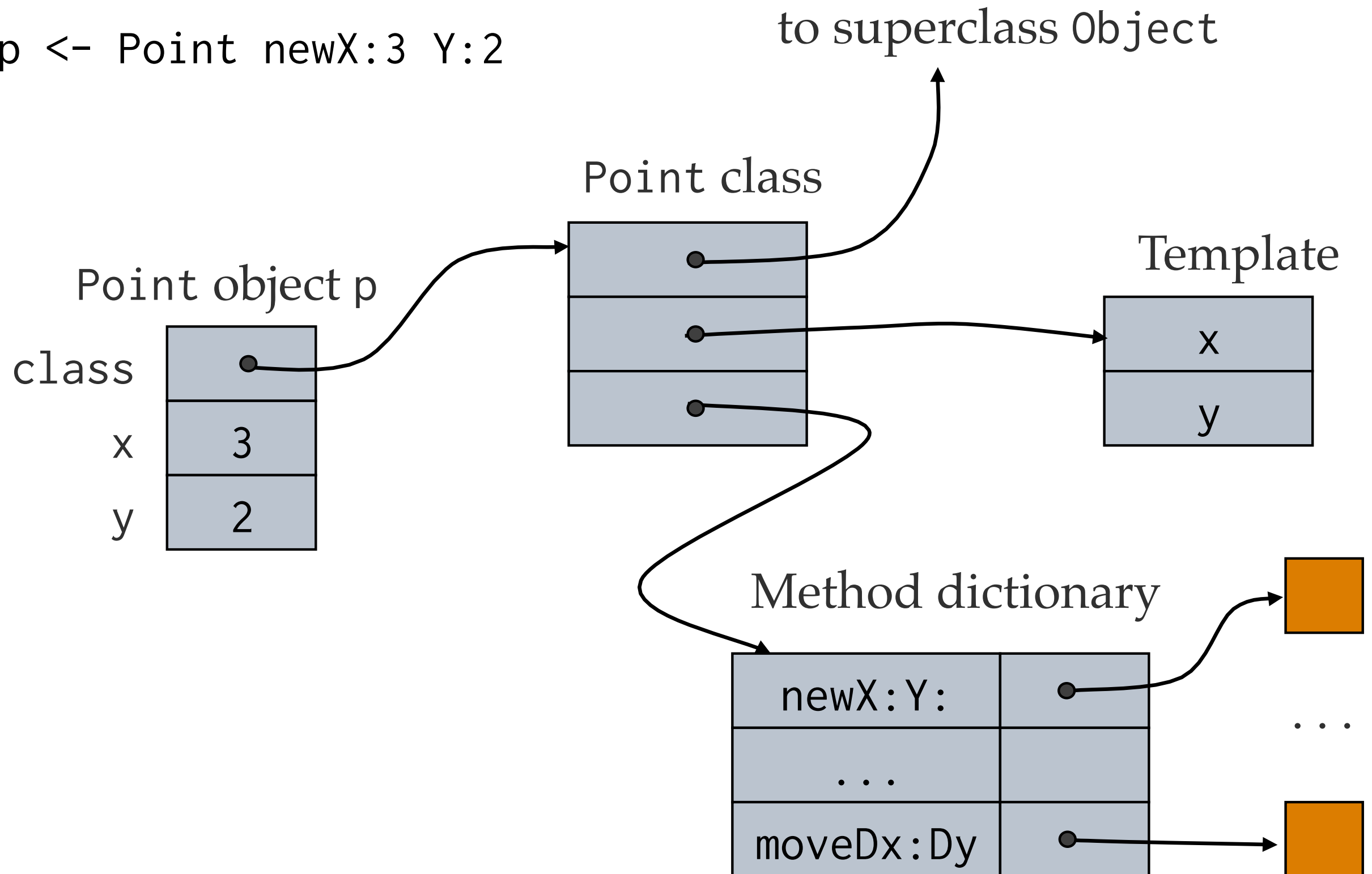
```
initialize ||  
pi <- 3.14159
```

How are objects represented?

- Objects have space for instance variable
- Objects have pointer to class
- Classes have pointers to
 - Super class (e.g., Object)
 - Template: names of instance variables
 - Method dictionary: maps selectors to code

Example representation of Point

```
p <- Point newX:3 Y:2
```



Example: Points

| | |
|-------------------------------|--------|
| class name | Point |
| super class | Object |
| class variables | pi |
| instance variables | x y |
| class messages and methods | |
| ⟨message & methods⟩ | |
| instance messages and methods | |
| ⟨message & methods⟩ | |

Example: Points

| | |
|-------------------------------|--------|
| class name | Point |
| super class | Object |
| class variables | pi |
| instance variables | x y |
| class messages and methods | |
| ⟨message & methods⟩ | |
| instance messages and methods | |
| ⟨message & methods⟩ | |

Inheritance

- Define ColoredPoint form Point:

| | |
|-------------------------------|--------------|
| class name | ColoredPoint |
| super class | Point |
| class variables | |
| instance variables | color |
| class messages and methods | |
| newX:xv Y:yv C:cv | ... |
| instance messages and methods | |
| color | ^ color |
| draw | ... |

Inheritance

- Define ColoredPoint form Point:

| | |
|-------------------------------|--------------|
| class name | ColoredPoint |
| super class | Point |
| class variables | |
| instance variables | color |
| class messages and methods | |
| newX:xv Y:yv C:cv | ... |
| instance messages and methods | |
| color | ^ color |
| draw | ... |

new instance variable



Inheritance

- Define ColoredPoint form Point:

| | |
|-------------------------------|--------------|
| class name | ColoredPoint |
| super class | Point |
| class variables | |
| instance variables | color |
| class messages and methods | |
| newX:xv Y:yv C:cv | ... |
| instance messages and methods | |
| color | ^ color |
| draw | ... |

new instance variable

new method

Inheritance

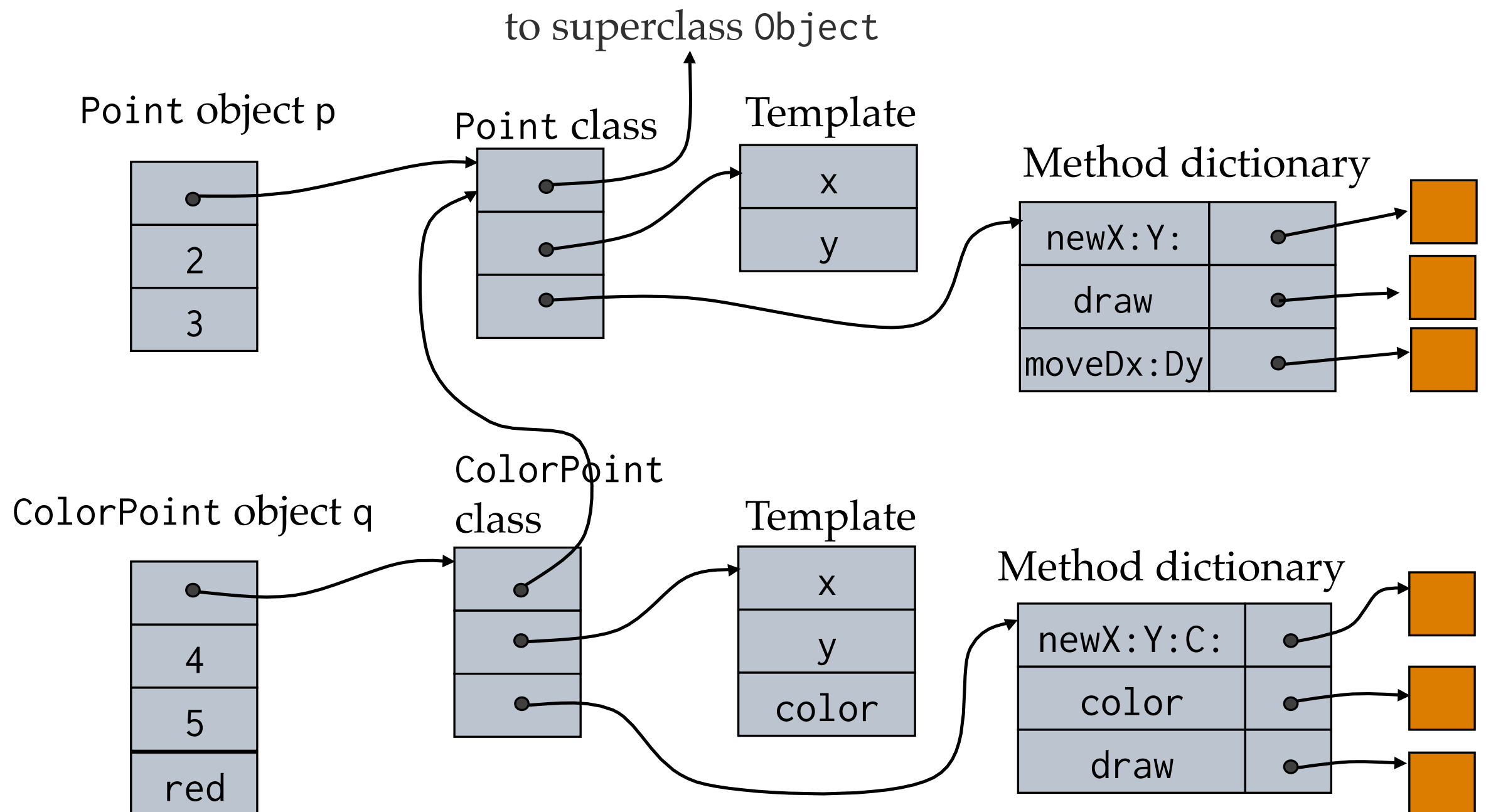
- Define ColoredPoint form Point:

| | | |
|-------------------------------|--------------|-----------------------|
| class name | ColoredPoint | |
| super class | Point | |
| class variables | | |
| instance variables | color | new instance variable |
| class messages and methods | | |
| newX:xv Y:yv C:cv | ... | new method |
| instance messages and methods | | |
| color | ^ color | override draw |
| draw | ... | method |

Run-time representation

```
p <- Point newX:3 Y:2
```

```
q <- ColorPoint newX:4 Y:5 C:red
```



What's the point?

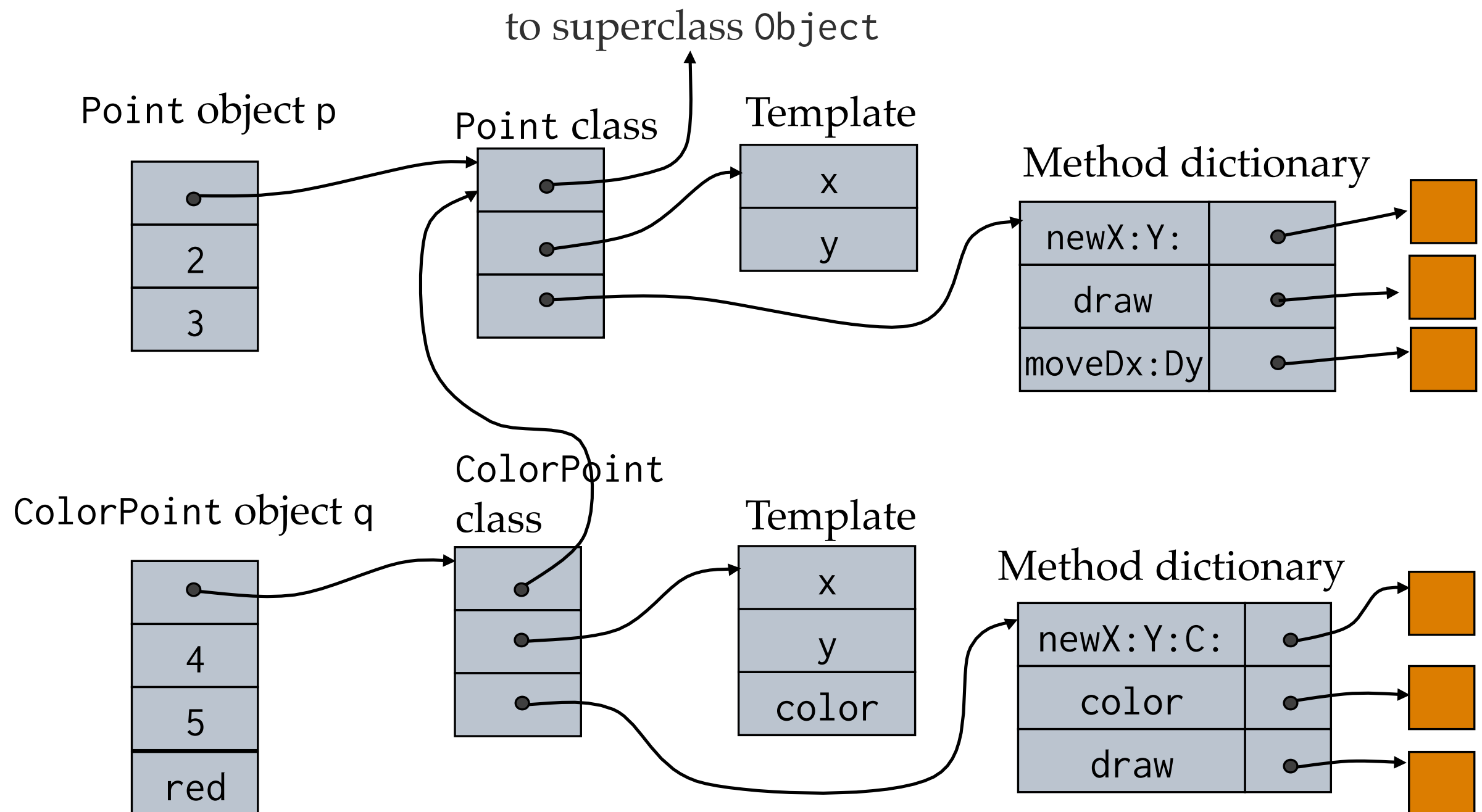
- Tells us exactly how to look up methods!
 - E.g., for Points: `p moveDx:5 Dy:5`
 - E.g., for ColorPoints: `q moveDx:5 Dy:5`

What's the point?

- Tells us exactly how to look up methods!
 - E.g., for Points: `p moveDx:5 Dy:5`
 - E.g., for ColorPoints: `q moveDx:5 Dy:5`

Dynamic lookup

- Dynamic lookup for p moveDx:5 Dy:5



What's the point?

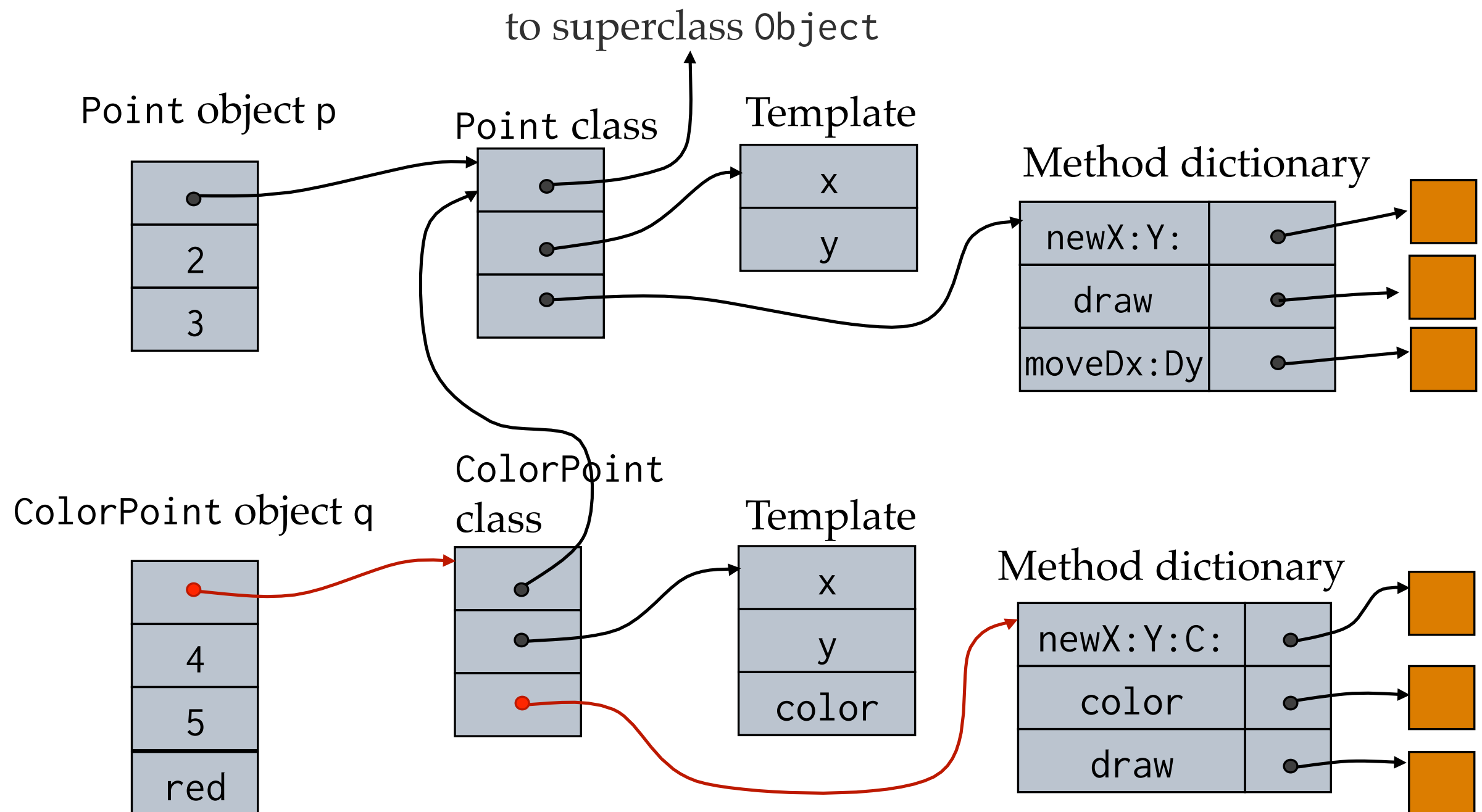
- Tells us exactly how to look up methods!
 - E.g., for Points: `p moveDx:5 Dy:5`
 - E.g., for ColorPoints: `q moveDx:5 Dy:5`

What's the point?

- Tells us exactly how to look up methods!
 - E.g., for Points: `p moveDx:5 Dy:5`
 - E.g., for ColorPoints: `q moveDx:5 Dy:5`

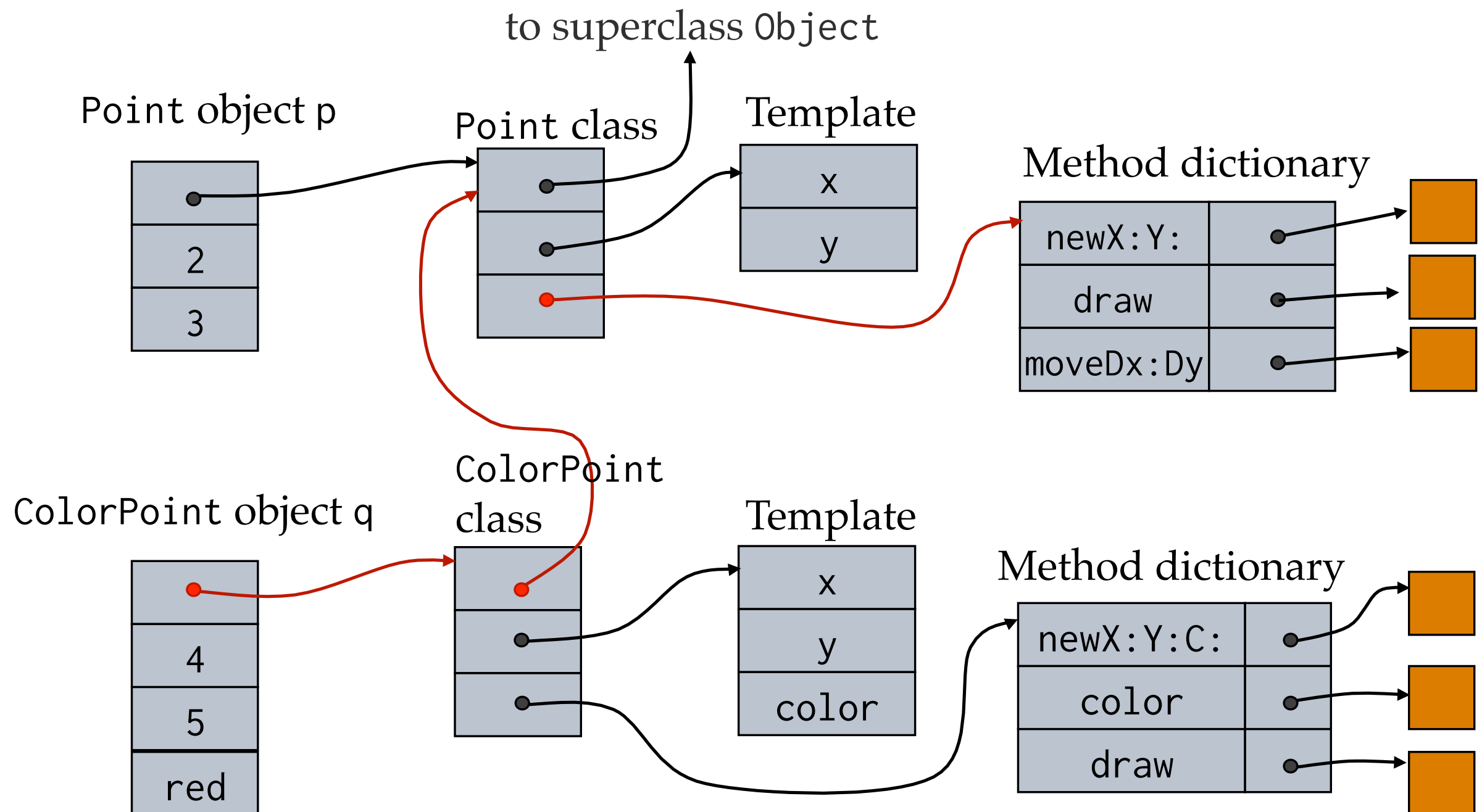
Dynamic lookup

- Dynamic lookup for q moveDx:5 Dy:5



Dynamic lookup

- What about q draw?



Smalltalk summary

- Classes: create objects that share methods
- Encapsulation: public methods, hidden instance vars
- Subtyping: implicit (based on handled messages)
- Inheritance: subclasses, self, super

Outline

- Central concepts in object-oriented languages
- Objects as activation records (Simula)
- Dynamically-typed object-oriented languages
 - Class-based languages (Smalltalk)
- ➔ Prototype-based languages (JavaScript)

JavaScript: the Self parts

Self

- Prototype-based pure object-oriented language
- Designed at Xerox PARC & Stanford
- Dynamically typed, everything is an object
- Operations on objects
 - send message, add new slot, replace old slot, remove slot
- No compelling application until JavaScript

JavaScript: the Self parts

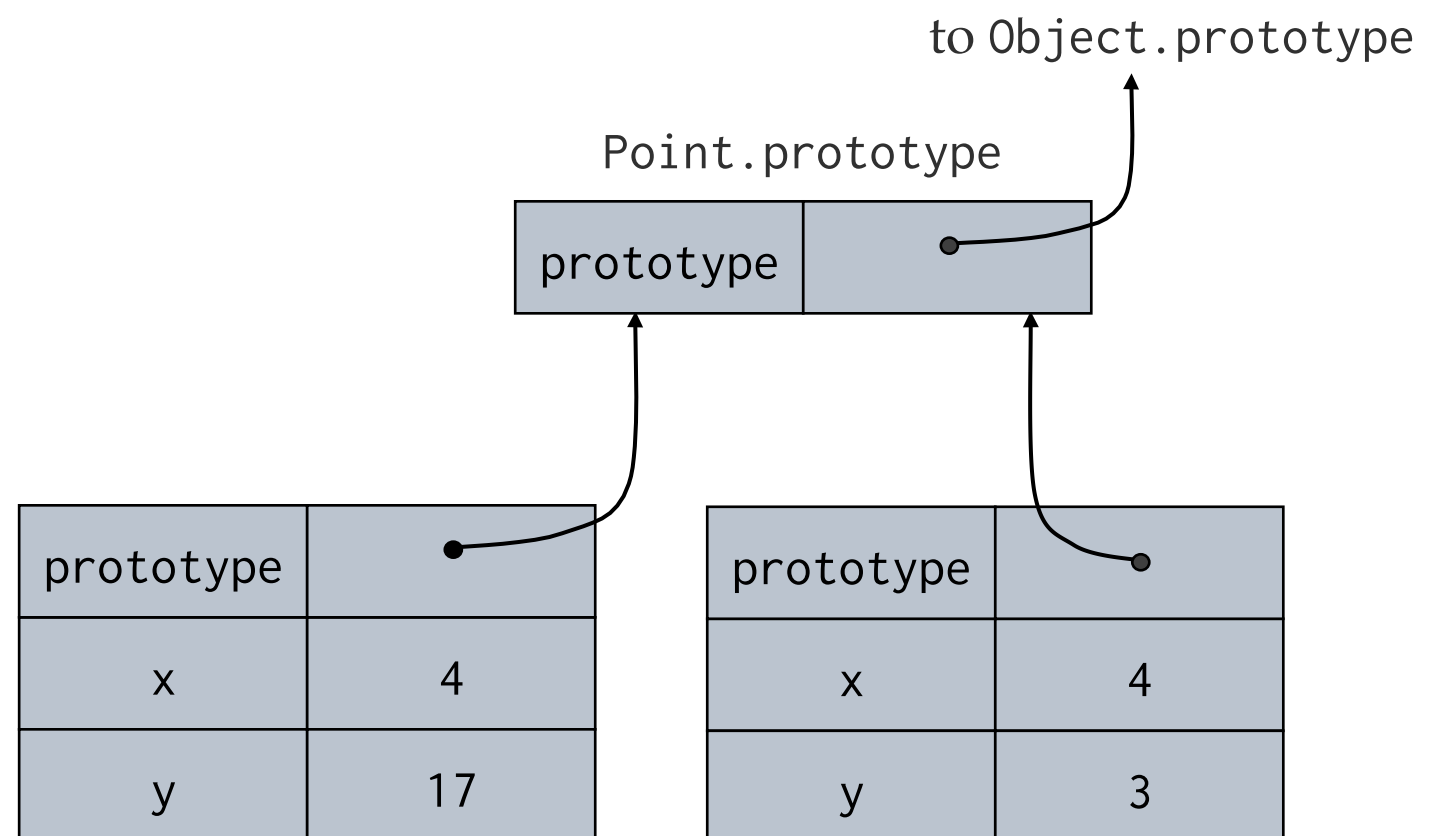
- Object is a collection of properties (named values)
 - Data properties are like “instance variables”
 - Retrieved by effectively sending **get message** to object
 - Assigned by effectively sending **set message** to object
 - Methods: properties containing JavaScript code
 - Have access to object of this method called **this**
 - Prototype (i.e., parent)
 - Points to existing object to inherit properties

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;  
}
```

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

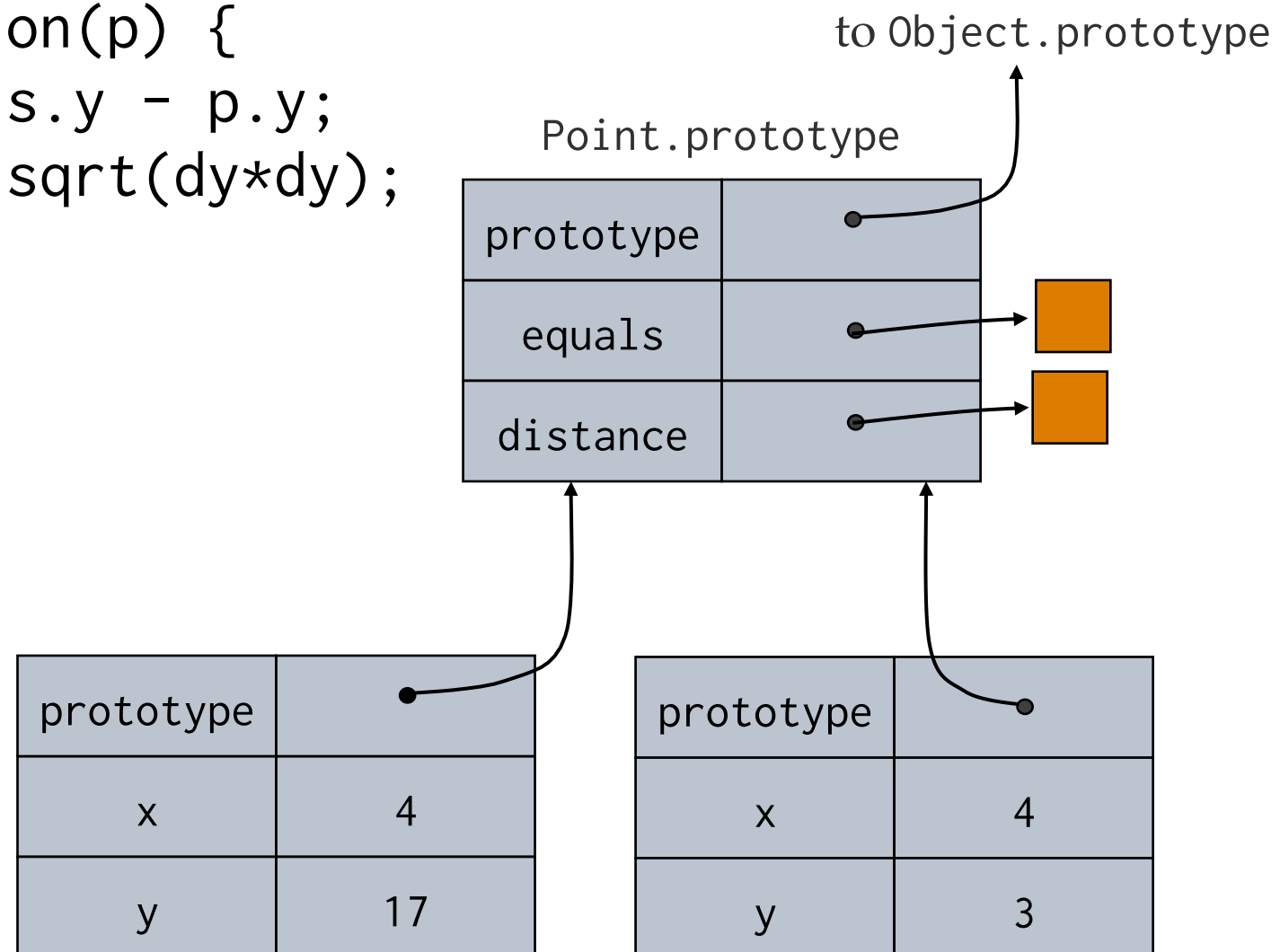


Inheriting properties

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

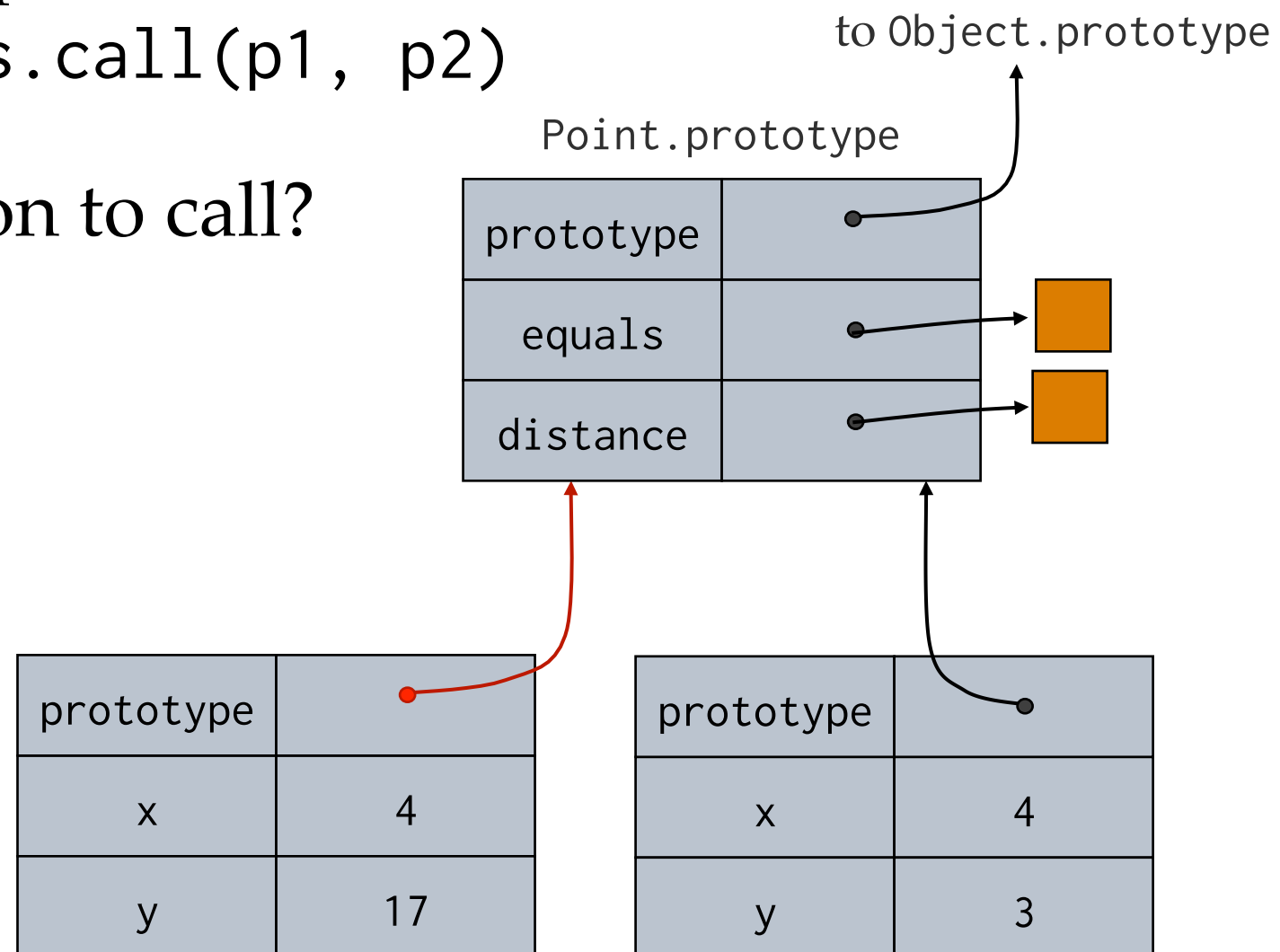
```
Point.prototype.distance = function(p) {  
  var dx = this.x - p.x, dy = this.y - p.y;  
  return Math.sqrt(dx*dx) + Math.sqrt(dy*dy);  
}
```

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



How does method invocation work?

- 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.toHashCode()`;

Dynamic lookup

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

Encapsulation & subtyping

- Encapsulation
 - Public methods
 - No private / protected data
 - Can use WeakMaps to do encapsulation, ugly
- Subtyping
 - Interface: the messages an object implements methods for
 - Solely need to define the right properties to have $<:$ relation

Inheritance

Inheritance

Let's make ColoredPoint inherit from Point:

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

Inheritance

Let's make ColoredPoint inherit from Point:

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

- Not exactly what we want! Why?

Inheritance

Let's make ColoredPoint inherit from Point:

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

- Not exactly what we want! Why?
- Suppose we wish to redefine equals for ColoredPoint

Inheritance

Let's make ColoredPoint inherit from Point:

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

- Not exactly what we want! Why?
- Suppose we wish to redefine equals for ColoredPoint
 - This would redefine equals for Point as well!

Inheritance

Let's make ColoredPoint inherit from Point:

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

- Not exactly what we want! Why?
- Suppose we wish to redefine equals for ColoredPoint
 - This would redefine equals for Point as well!
- Solution:

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

Inheritance

Let's make ColoredPoint inherit from Point:

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

- Not exactly what we want! Why?
- Suppose we wish to redefine equals for ColoredPoint
 - This would redefine equals for Point as well!
- Solution:

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

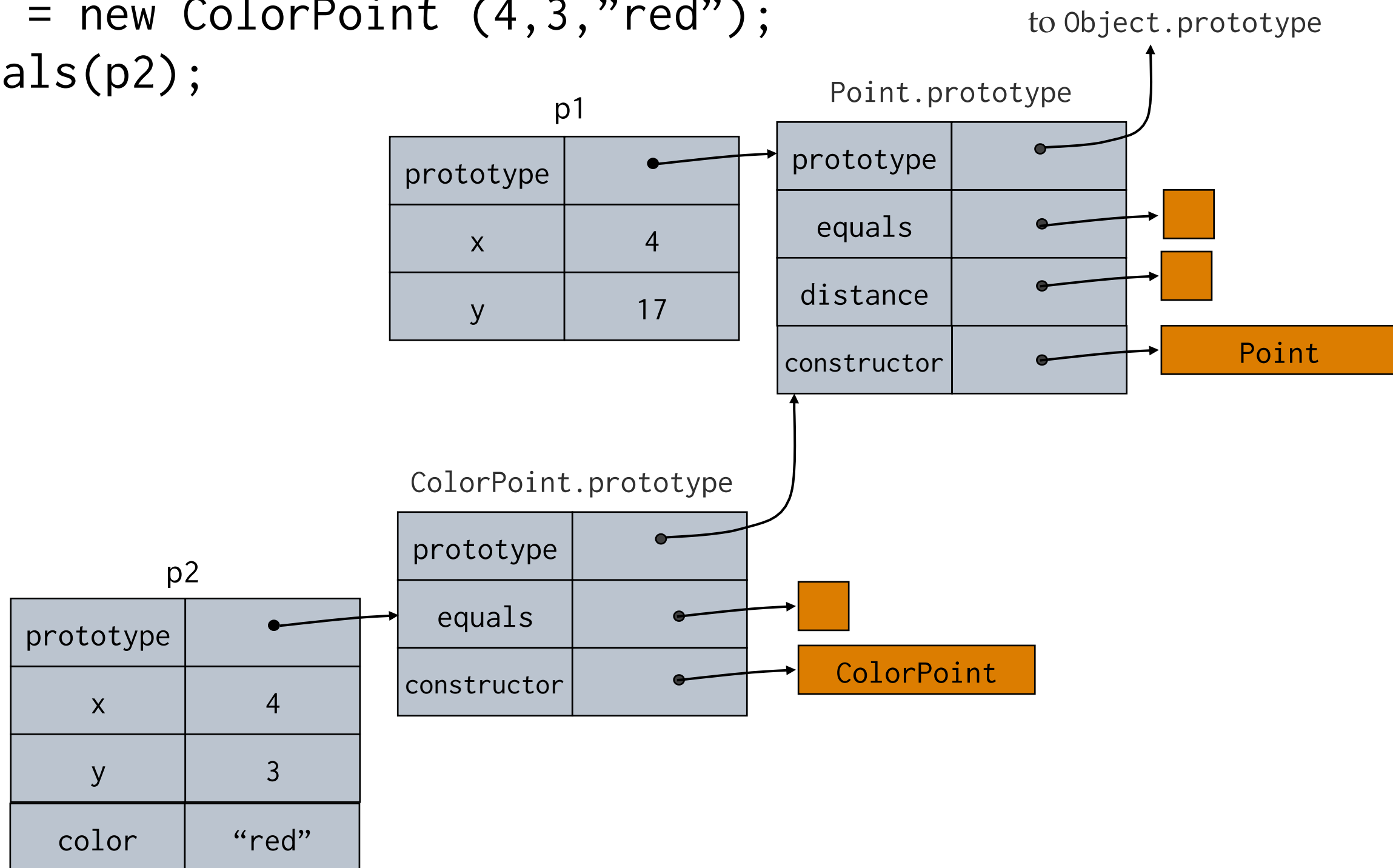
- Object.create creates new object with specified prototype

Inheritance

```
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

```
var p1 = new Point (4,17);  
var p2 = new ColorPoint (4,3,"red");  
p1.equals(p2);
```



JavaScript summary

- Objects: created by calling functions as constructors
- Encapsulation: public methods, hidden instance vars
- Subtyping: implicit (based on handled messages)
- Inheritance: prototype hierarchy
- Classes: desugars to prototypal implementation

Outline

- Central concepts in object-oriented languages
 - Dynamic lookup, encapsulation, subtyping, inheritance
- Objects as activation records (Simula)
- Dynamically-typed object-oriented languages
 - Class-based languages (Smalltalk)
 - Prototype-based languages (JavaScript)