

Objects, Encapsulation, Inheritance

Reading (two lectures)

Chapter 10, except section 10.4

Chapter 11, sections 11.1, 11.2, 11.3.1 and 11.4., 11.5, 11.6 only

Chapter 12, sections 12.1, 12.2, 12.3 and 12.5 only

Chapter 13, sections 13.1 and 13.2 only

Warning

- The lecture organization doesn't exactly match the book organization
 - The book covers selected object-oriented languages in historical order
 - Lectures are organized more by concept instead of by language

Outline

- Central concepts in object-oriented languages
 - Dynamic lookup, encapsulation, subtyping, inheritance
- Objects as activation records
 - Simula: implementation as activation records with static scope
- Pure dynamically-typed object-oriented languages
 - Object implementation and run-time lookup
 - Class-based languages (Smalltalk)
 - Prototype-based languages (Self, JavaScript)
- Statically-typed object-oriented languages (second lecture)
 - C++ – using static typing to eliminate search
 - problems with C++ multiple inheritance
 - Java – using Interfaces to avoid multiple inheritance

Object-oriented programming

- Primary object-oriented language concepts
 - dynamic lookup
 - encapsulation
 - inheritance
 - subtyping
- Program organization
 - Work queue, geometry program, design patterns
- Comparison
 - Objects as closures?

Objects

- An object consists of
 - hidden data
 - instance variables, also called fields, data members, ...
 - hidden functions also possible
 - public operations
 - methods or member functions
 - can also have public variables in some languages
- Object-oriented program:
 - Send messages to objects

hidden data	
msg ₁	method ₁
...	...
msg _n	method _n

What's interesting about this?

- Universal encapsulation construct
 - Data structure
 - File system
 - Database
 - Window
 - Integer
- Metaphor usefully ambiguous
 - sequential or concurrent computation
 - distributed, sync. or async. communication

Object-Orientation

- Programming methodology
 - organize concepts into objects and classes
 - build extensible systems
- Language concepts
 - dynamic lookup
 - encapsulation
 - subtyping allows extensions of concepts
 - inheritance allows reuse of implementation

Dynamic Lookup

- In object-oriented programming,
object → message (arguments)
code depends on object and message
- In conventional programming,
operation (operands)
meaning of operation is always the same

Fundamental difference between abstract data types (alone) and objects

Example

- Add two numbers $x \rightarrow \text{add}(y)$
different **add** if **x** is integer, string
- Conventional programming **add(x, y)**
function **add** has fixed meaning

Important distinction:

Overloading is resolved at compile time

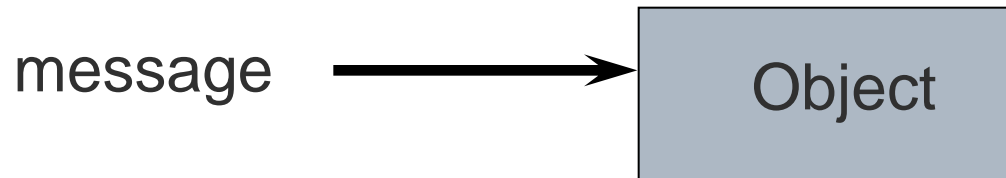
Dynamic lookup is a run time operation

Language concepts

- “dynamic lookup”
 - different code for different objects
 - integer “+” different from string “+”
- encapsulation
- subtyping
- inheritance

Encapsulation

- Builder of a concept has detailed view
- User of a concept has “abstract” view
- Encapsulation separates these two views
 - Implementation code: operate on representation
 - Client code: operate by applying fixed set of operations provided by implementer of abstraction



Language concepts

- “Dynamic lookup”
 - different code for different object
 - integer “+” different from real “+”
- Encapsulation
 - Implementer of a concept has detailed view
 - User has “abstract” view
 - Encapsulation separates these two views
- Subtyping
- Inheritance

Subtyping and Inheritance

- **Interface**
 - The external view of an object
- **Subtyping**
 - Relation between interfaces
- **Implementation**
 - The internal representation of an object
- **Inheritance**
 - Relation between implementations

Object Interfaces

- Interface
 - The messages understood by an object
- Example: point
 - `x-coord` : returns x-coordinate of a point
 - `y-coord` : returns y-coordinate of a point
 - `move` : method for changing location
- The interface of an object is its *type*

Subtyping

- If interface **A** contains all of interface **B**, then **A** objects can also be used **B** objects.

Point

x-coord
y-coord
move

Colored_point

x-coord
y-coord
color
move
change_color

Colored_point interface contains Point
Colored_point is a subtype of Point

Inheritance

- Implementation mechanism
- New objects may be defined by reusing implementations of other objects

Example

```
class Point
```

```
    private
```

```
        float x, y
```

```
    public
```

```
        point move (float dx, float dy);
```

```
class Colored_point
```

```
    private
```

```
        float x, y; color c
```

```
    public
```

```
        point move(float dx, float dy);
```

```
        point change_color(color newc);
```

- Subtyping

- Colored points can be used in place of points
- Property used by client program

- Inheritance

- Colored points can be implemented by reusing point implementation
- Technique used by implementer of classes

OO Program Structure

- Group data and functions
- Class
 - Defines behavior of all objects that are instances of the class
- Subtyping
 - Place similar data in related classes
- Inheritance
 - Avoid reimplementing functions that are already defined

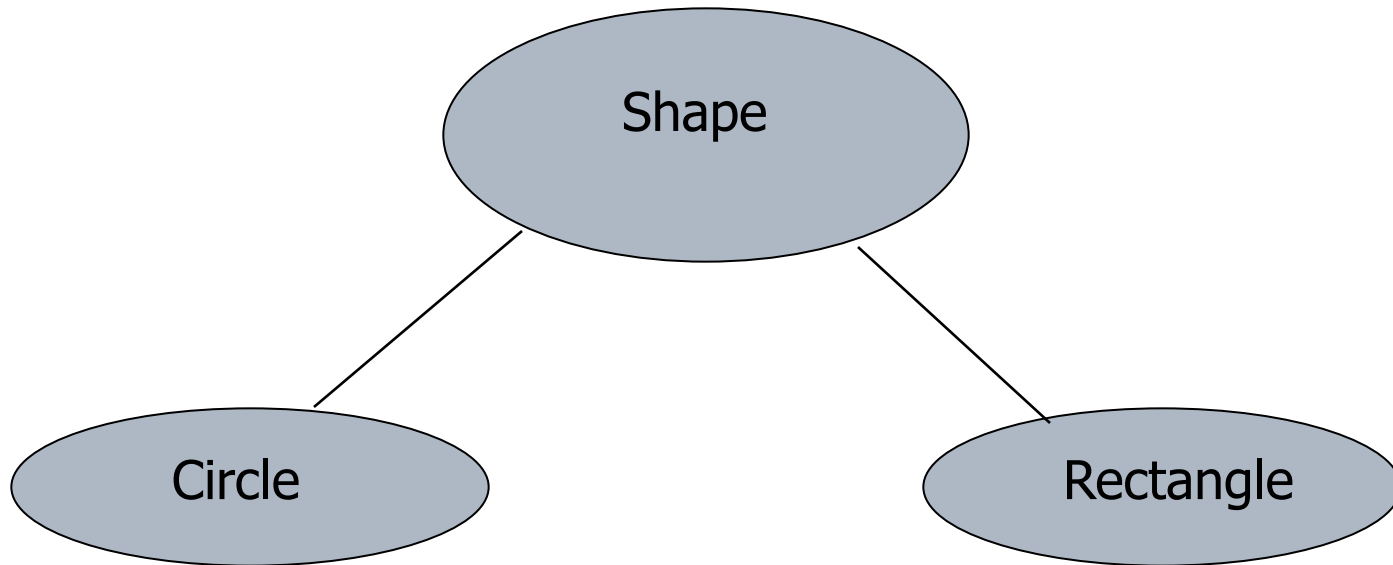
Example: Geometry Library

- Define general concept: **shape**
- Implement two shapes: **circle, rectangle**
- Functions on implemented shapes
center, move, rotate, print
- Anticipate additions to library

Shapes

- Interface of every **shape** must include
center, move, rotate, print
- Different kinds of shapes are implemented differently
 - **Square**: four points, representing corners
 - **Circle**: center point and radius

Subtype hierarchy



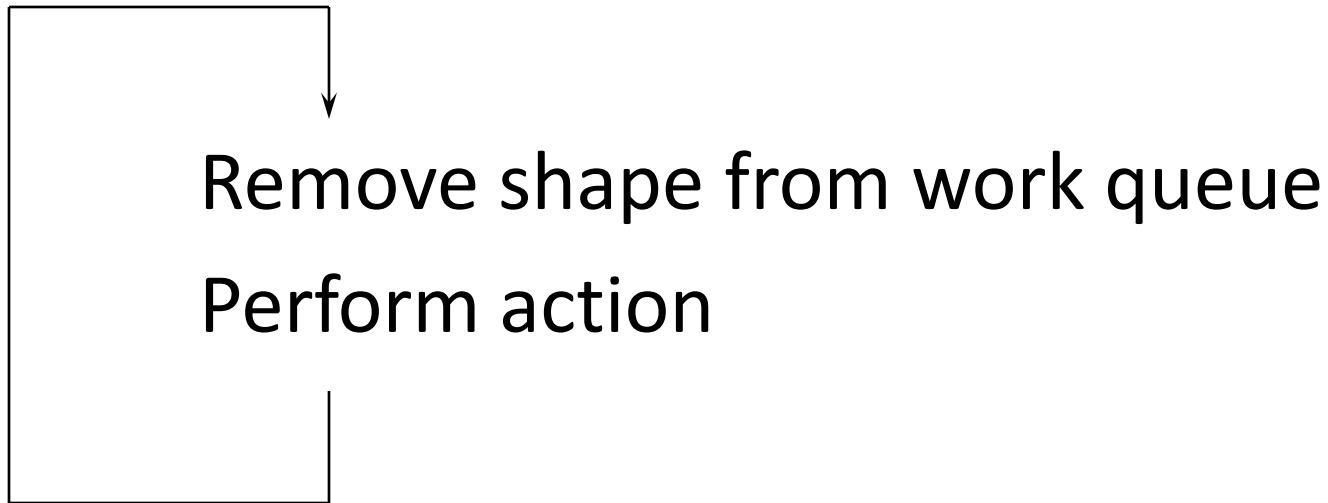
- General interface defined in the **shape** class
- Implementations defined in **circle, rectangle**
- Extend hierarchy with additional shapes

Code placed in classes

	center	move	rotate	print
Circle	c_center	c_move	c_rotate	c_print
Rectangle	r_center	r_move	r_rotate	r_print

- Dynamic lookup
circle → move(x,y) calls function c_move
- Conventional organization
Place c_move, r_move in move function

Example use: Processing Loop



- Control loop does not know the type of each shape

Outline

- Central concepts in object-oriented languages
 - Dynamic lookup, encapsulation, subtyping, inheritance
- ➔ **Objects as activation records**
 - Simula – implementation as activation records with static scope
- **Pure dynamically-typed object-oriented languages**
 - Object implementation and run-time lookup
 - Class-based languages (Smalltalk)
 - Prototype-based languages (Self, JavaScript)
- **Statically-typed object-oriented languages**
 - C++ – using static typing to eliminate search
 - problems with C++ multiple inheritance
 - Java – using Interfaces to avoid multiple inheritance

Simula: objects as activation records

- Simula 67: First object-oriented language
- Designed for simulation
 - Later recognized as general-purpose prog language
- Extension of Algol 60
- Standardized as Simula (no “67”) in 1977
- Inspiration to many later designers
 - Smalltalk
 - C++
 - ...

Brief history

- Norwegian Computing Center
 - Designers: Dahl, Myhrhaug, Nygaard
 - Simula-1 in 1966 (strictly a simulation language)
 - General language ideas
 - Influenced by Hoare's ideas on data types
 - Added classes and prefixing (subtyping) to Algol 60
 - Nygaard
 - Operations Research specialist and political activist
 - Wanted language to describe social and industrial systems
 - Allow "ordinary people" to understand political (?) changes
 - Dahl and Myhrhaug
 - Maintained concern for general programming

Objects in Simula

- **Class**
 - A procedure that returns a pointer to its activation record
- **Object**
 - Activation record produced by call to a class
- **Object access**
 - Access any local variable or procedures using dot notation: **object.var**
- **Memory management**
 - Objects are garbage collected
 - user destructors considered undesirable

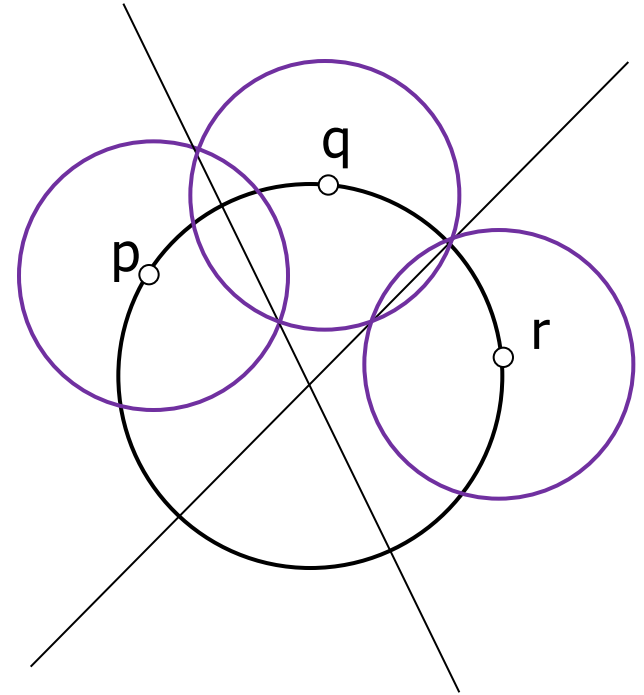
Example: Circles and lines

- **Problem**

- Find the center and radius of the circle passing through three distinct points, p , q , and r

- **Solution**

- Draw intersecting circles C_p , C_q around p, q and circles $C_{q'}$, C_r around q, r (Picture assumes $C_q = C_{q'}$)
- Draw lines through circle intersections
- The intersection of the lines is the center of the desired circle.
- Error if the points are colinear.



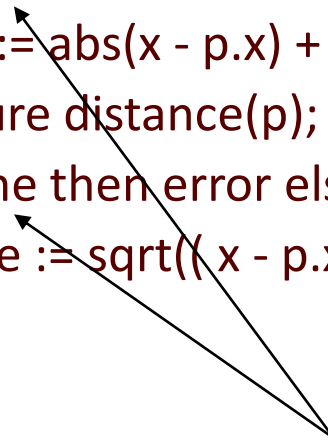
Approach in Simula

- Methodology
 - Represent points, lines, and circles as objects.
 - Equip objects with necessary operations.
- Operations
 - Point
 - equality(anotherPoint) : boolean
 - distance(anotherPoint) : real (needed to construct circles)
 - Line
 - parallelto(anotherLine) : boolean (to see if lines intersect)
 - meets(anotherLine) : REF(Point)
 - Circle
 - intersects(anotherCircle) : REF(Line)

Simula Point Class

```
class Point(x,y); real x,y;
begin
  boolean procedure equals(p); ref(Point) p;
  if p /= none then
    equals := abs(x - p.x) + abs(y - p.y) < 0.00001
  real procedure distance(p); ref(Point) p;
  if p == none then error else
    distance := sqrt(( x - p.x )**2 + (y - p.y) ** 2);
end ***Point***
```

formal p is pointer to Point

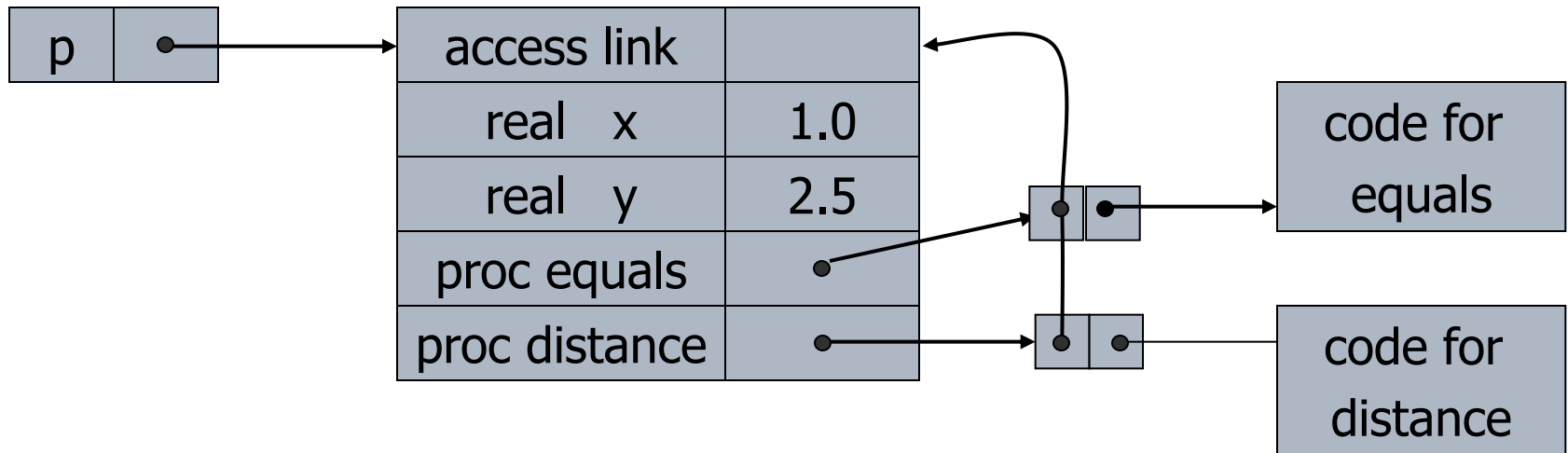


```
p :- new Point(1.0, 2.5);
q :- new Point(2.0,3.5);
if p.distance(q) > 2 then ...
```

uninitialized ptr has
value none

pointer assignment

Representation of objects



Object is represented by activation record with access link to find global variables according to static scoping

Simula line class

```
class Line(a,b,c); real a,b,c;
begin
  boolean procedure parallelto(l); ref(Line) l;
    if l /= none then parallelto := ...
  ref(Point) procedure meets(l); ref(Line) l;
    begin real t;
      if l /= none and ~parallelto(l) then ...
    end;
  real d; d := sqrt(a**2 + b**2);
  if d = 0.0 then error else
    begin
      d := 1/d;
      a := a*d; b := b*d; c := c*d;
    end;
end *** Line***
```

← Local variables

line determined by
 $ax+by+c=0$

Procedures

Initialization:
"normalize" a,b,c

Derived classes in Simula

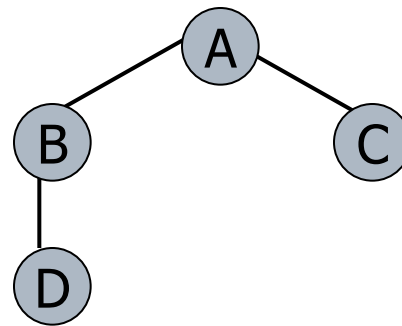
- A class decl may be prefixed by a class name

class A

A class B

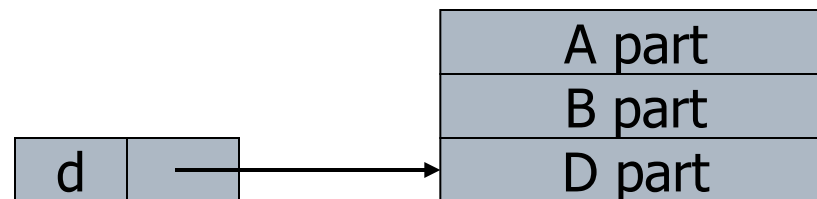
A class C

B class D



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

d :- new D(...)



Main object-oriented features

- Classes
- Objects
- Inheritance (“class prefixing”)
- Subtyping
- Virtual methods
 - A function can be redefined in subclass
- Inner
 - Combines code of superclass with code of subclass
- Inspect/Qua
 - run-time class/type tests


Features absent from Simula 67

- Encapsulation
 - All data and functions accessible; no private, protected
- Self/Super mechanism of Smalltalk
 - But has an expression `this<class>` to refer to object itself, regarded as object of type `<class>`. Not clear how powerful this is...
- Class variables
 - But can have global variables
- Exceptions
 - Not fundamentally an OO feature ...

Simula Summary

- **Class**
 - "procedure" that returns ptr to activation record
 - initialization code always run as procedure body
- **Objects:** closure created by a class
- **Encapsulation**
 - protected and private not recognized in 1967
 - added later and used as basis for C++
- **Subtyping:** determined by class hierarchy
- **Inheritance:** provided by class prefixing

Outline

- Central concepts in object-oriented languages
 - Dynamic lookup, encapsulation, subtyping, inheritance
- Objects as activation records
 - Simula – implementation as activation records with static scope
-  Pure dynamically-typed object-oriented languages
 - Object implementation and run-time lookup
 - Class-based languages (Smalltalk)
 - Prototype-based languages (Self, JavaScript)
- Statically-typed object-oriented languages
 - C++ – using static typing to eliminate search
 - problems with C++ multiple inheritance
 - Java – using Interfaces to avoid multiple inheritance

Smalltalk

- Major language that popularized objects
- Developed at Xerox PARC
 - Smalltalk-76, Smalltalk-80 were important versions
- Object metaphor extended and refined
 - Used some ideas from Simula, but very different lang
 - Everything is an object, even a class
 - All operations are “messages to objects”
 - Very flexible and powerful language
 - Similar to “everything is a list” in Lisp, but more so
 - Example: object can detect that it has received a message it does not understand, can try to figure out how to respond.

Motivating application: Dynabook

- Concept developed by Alan Kay
- Small portable computer
 - Revolutionary idea in early 1970's
 - At the time, a *minicomputer* was shared by 10 people, stored in a machine room.
 - What would you compute on an airplane?
- Influence on Smalltalk
 - Language intended to be programming language and operating system interface
 - Intended for “non-programmer”
 - Syntax presented by language-specific editor

Smalltalk language terminology

- **Object** Instance of some class
- **Class** Defines behavior of its objects
- **Selector** Name of a message
- **Message** Selector together with parameter values
- **Method** Code used by a class to respond to message
- **Instance variable** Data stored in object
- **Subclass** Class defined by giving incremental modifications to some superclass

Example: Point class

- Class definition written in tabular form

class name	Point
super class	Object
class var	pi
instance var	x y
class messages and methods	
⟨...names and code for methods...⟩	
instance messages and methods	
⟨...names and code for methods...⟩	

Class messages and methods

Three class methods

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

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

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

- **Explanation**

- selector is mix-fix newX:Y:
e.g, Point newX:3 Y:2
- symbol ^ marks return value
- new is method in all classes,
inherited from Object
- | | marks scope for local decl

- initialize method sets pi, called
automatically
- <- is syntax for assignment

Instance messages and methods

Five instance methods

```
x: xcoord y: ycoord | |
```

```
  x <- xcoord
```

```
  y <- ycoord
```

```
moveDx: dx Dy: dy | |
```

```
  x <- dx + x
```

```
  y <- dy + y
```

```
x | | ^x
```

```
y | | ^y
```

```
draw | |
```

```
  <...code to draw point...>
```

- Explanation

set x,y coordinates,

e.g, pt x:5 y:3

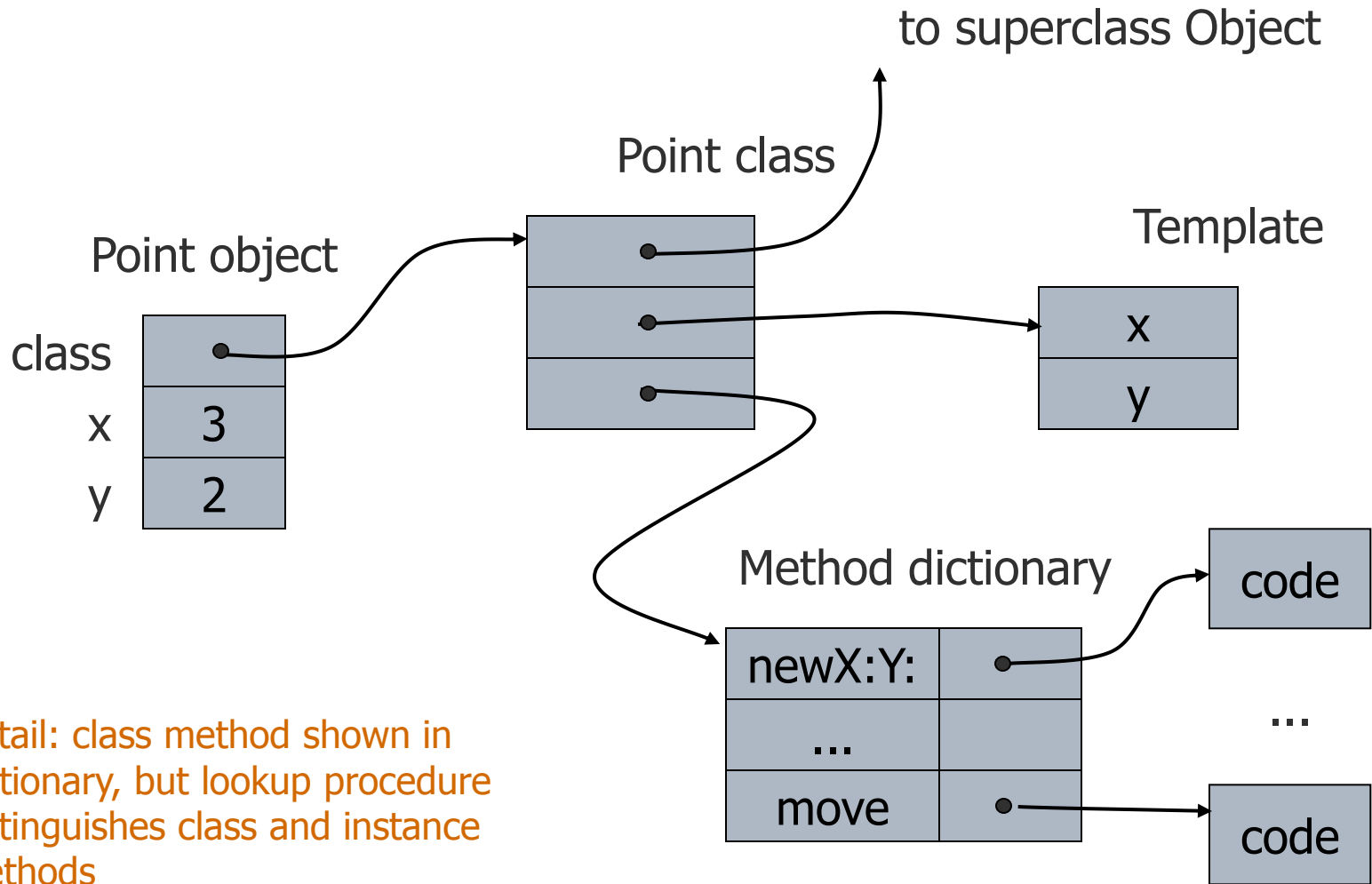
move point by given amount

return hidden inst var x

return hidden inst var y

draw point on screen

Run-time representation of point



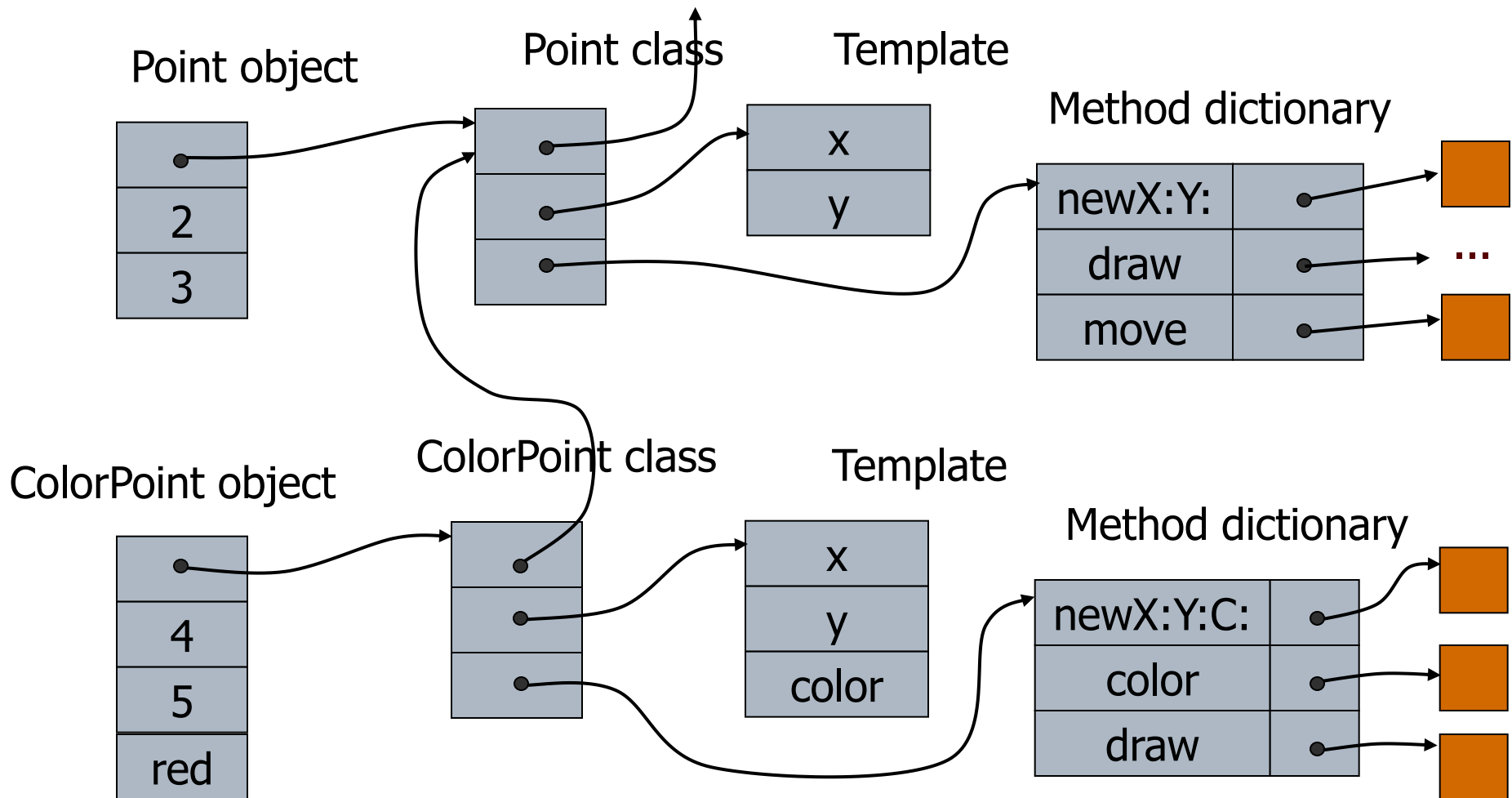
Detail: class method shown in dictionary, but lookup procedure distinguishes class and instance methods

Inheritance

- Define colored points from points

class name	ColorPoint	
super class	Point	
class var		
instance var	color	new instance variable
class messages and methods		
newX:xv Y:yv C:cv	< ... code ... >	new method
instance messages and methods		
color	^color	override Point method
draw	< ... code ... >	

Run-time representation



This is a schematic diagram meant to illustrate the main idea. Actual implementations may differ.

Encapsulation in Smalltalk

- Methods are public
- Instance variables are hidden
 - Not visible to other objects
 - `pt x` is not allowed unless `x` is a method
 - But may be manipulated by subclass methods
 - This limits ability to establish invariants
 - Example:
 - Superclass maintains sorted list of messages with some selector, say `insert`
 - Subclass may access this list directly, rearrange order

Smalltalk Summary

- **Class**
 - creates objects that share methods
 - pointers to template, dictionary, parent class
- **Objects:** created by a class, contains instance variables
- **Encapsulation**
 - methods public, instance variables hidden
- **Subtyping:** implicit, no static type system
- **Inheritance:** subclasses, self, super
 - Single inheritance in Smalltalk-76, Smalltalk-80

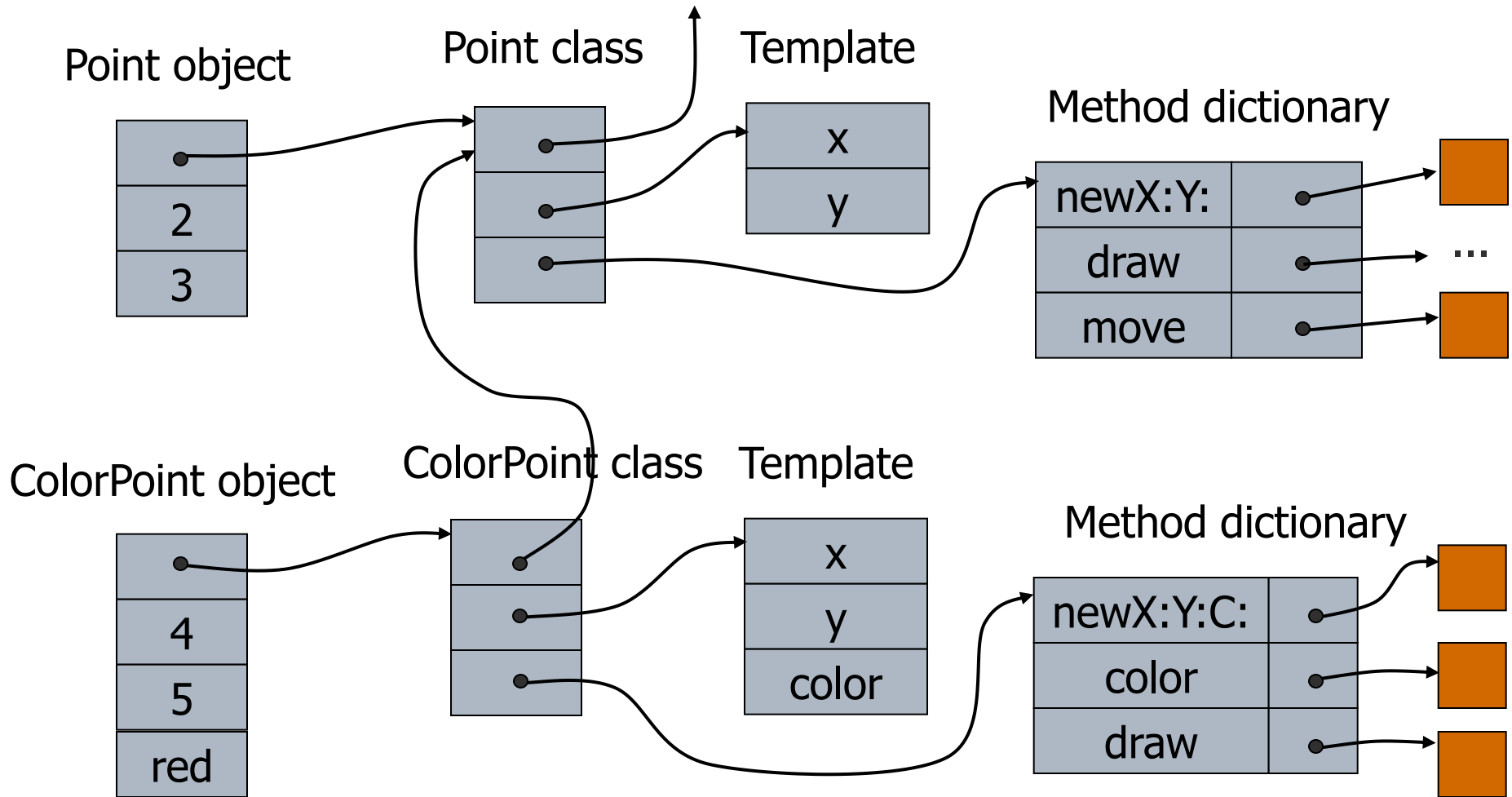
Self programming language

- Prototype-based pure object-oriented language.
- Designed by Randall Smith (Xerox PARC) and David Ungar (Stanford University)
 - Successor to Smalltalk-80
 - “Self: The power of simplicity” appeared at OOPSLA ‘87
 - Initial implementation done at Stanford; then project shifted to Sun Microsystems Labs
 - Vehicle for implementation research
- Self 4.3 available from Oracle web site:
<http://labs.oracle.com/self/>

Design Goals

- **Conceptual economy**
 - Everything is an object
 - Everything done using messages
 - No classes
 - No variables
- **Concreteness**
 - Objects should seem “real”
 - GUI to manipulate objects directly

“A Language for Smalltalk runtime structures”



How successful?

- Self is a carefully designed language
- Few users: not a popular success
 - No compelling application, until JavaScript
 - Influenced development of object calculi w/o classes
- However, many research innovations
 - Very simple computational model
 - Enormous advances in compilation techniques
 - Influenced the design of Java compilers

Language Overview

- Dynamically typed
- Everything is an object
- All computation via message passing
- Creation and initialization: clone object
- Operations on objects:
 - send messages
 - add new slots
 - replace old slots
 - remove slots

Objects and Slots

Object consists of named slots.

– Data

- Such slots return contents upon evaluation; so act like instance variables

– Assignment

- Set the value of associated slot

– Method

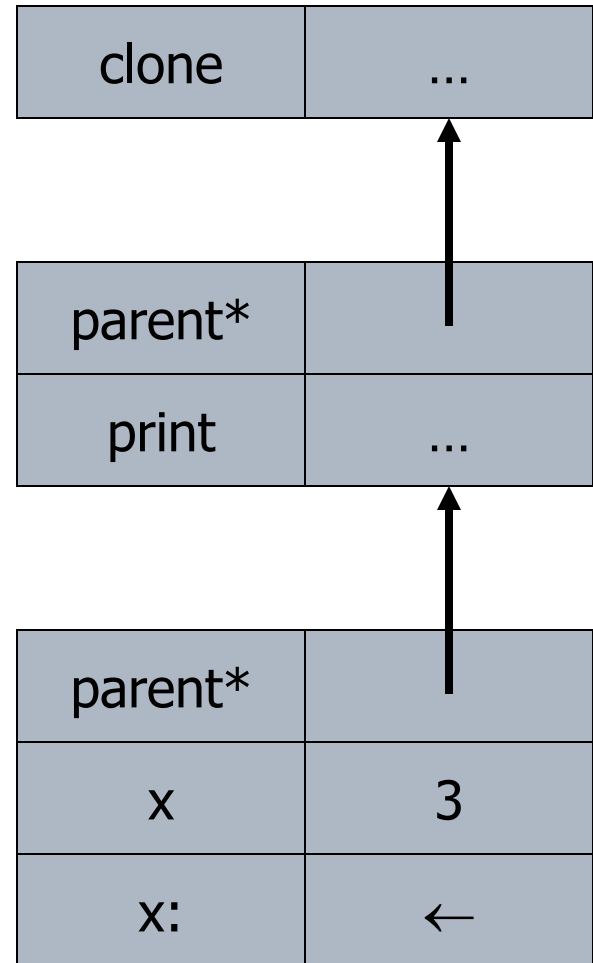
- Slot contains Self code

– Parent

- Point to existing object to inherit slots

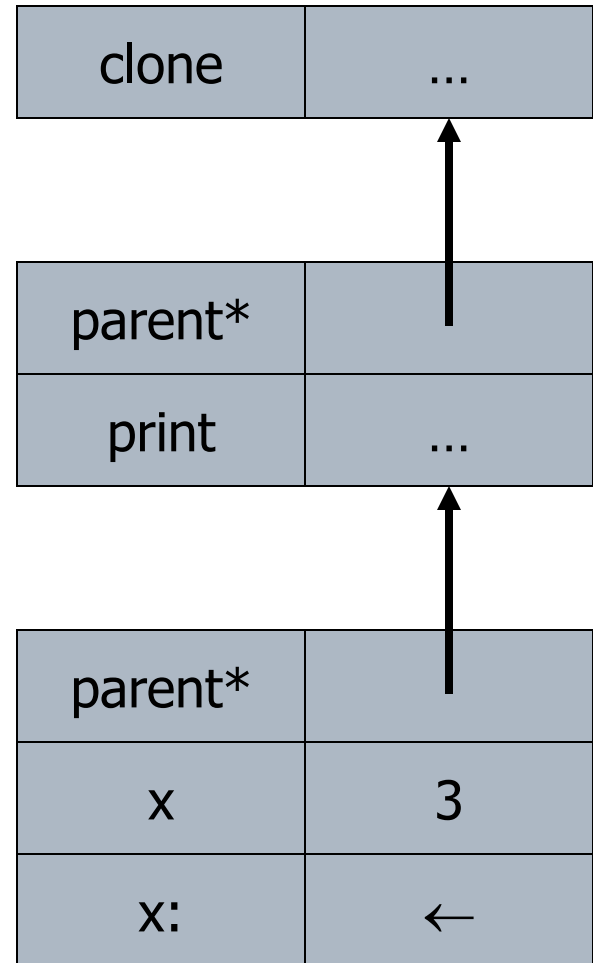
Messages and Methods

- When message is sent, object searched for slot with name.
- If none found, all parents are searched.
 - Runtime error if more than one parent has a slot with the same name.
- If slot is found, its contents evaluated and returned.
 - Runtime error if no slot found.

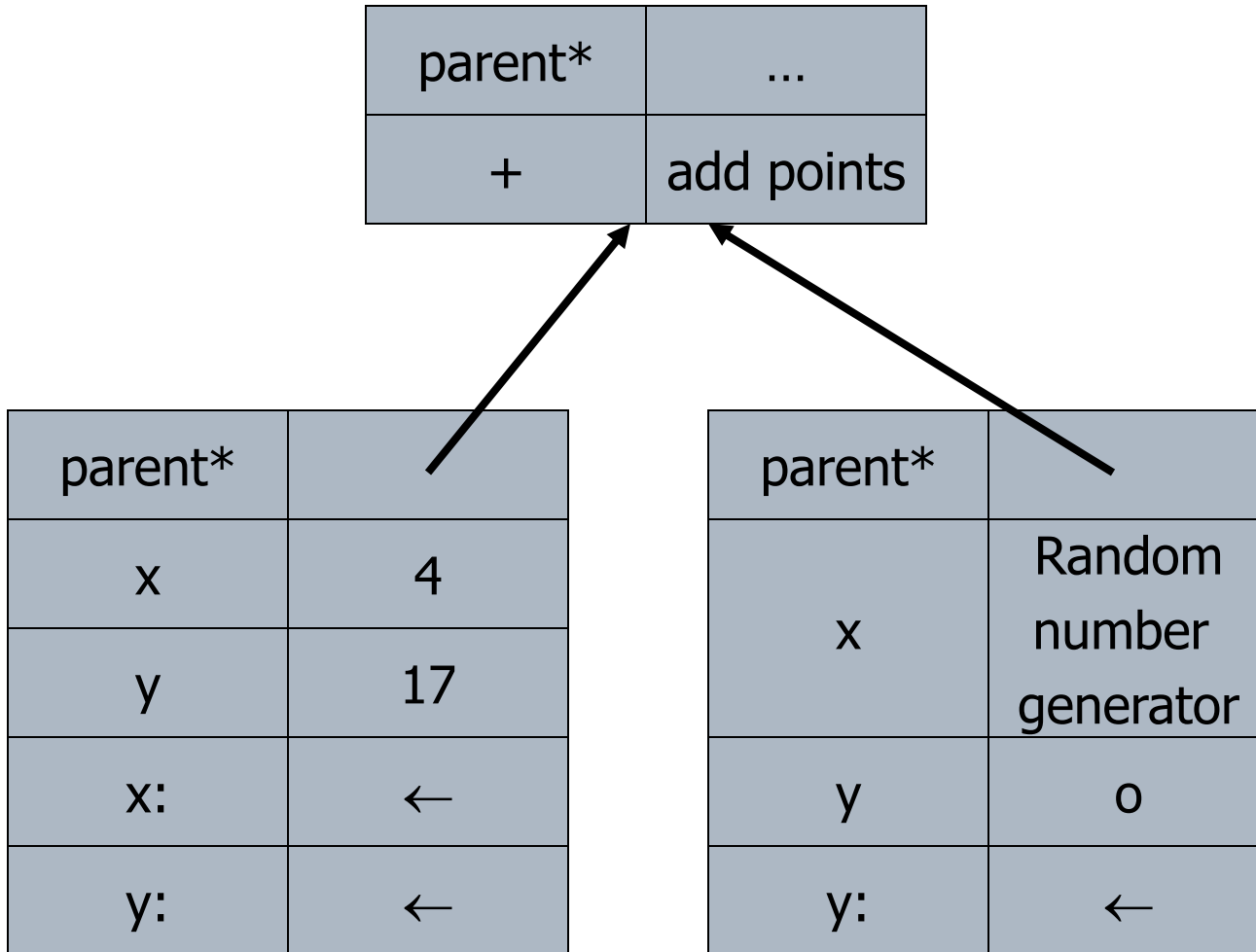


Messages and Methods

obj x 3
obj print *print point object*
obj x: 4 *obj after setting x to 4*



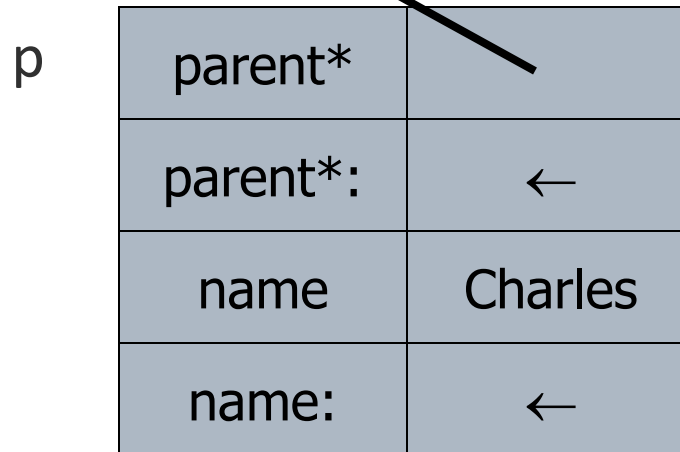
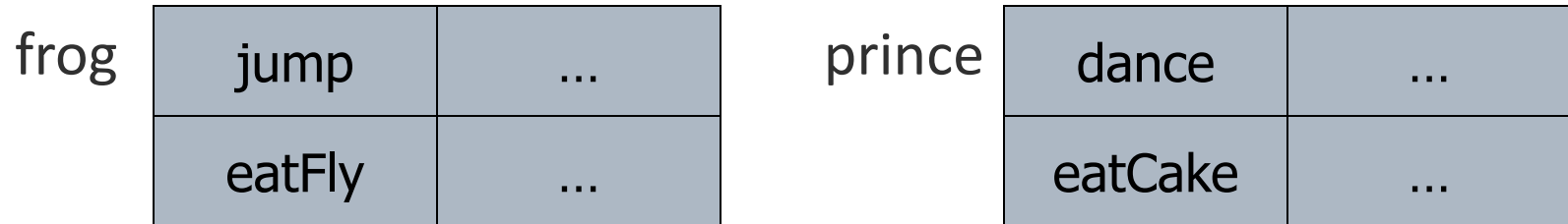
Mixing State and Behavior



Object Creation

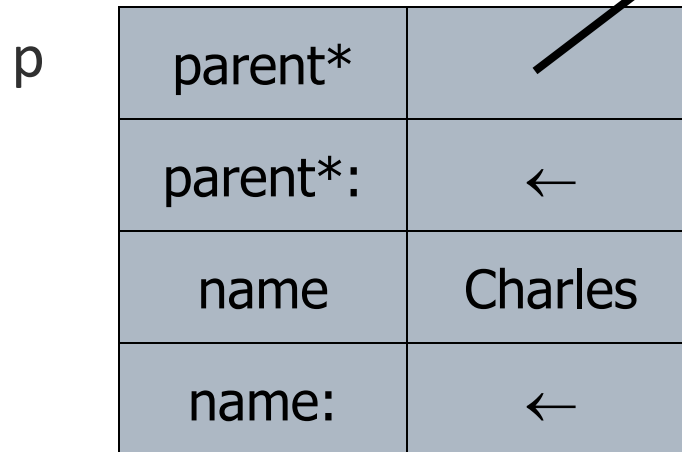
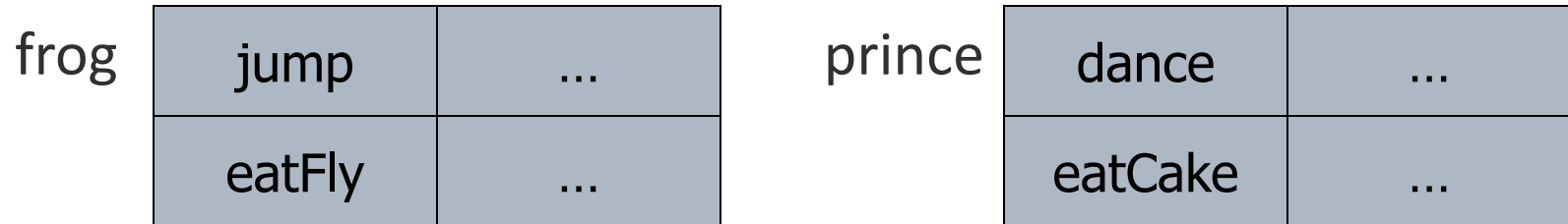
- To create an object, we copy an old one
- We can **add** new methods, **override** existing ones, or even **remove** methods
- These operations also apply to **parent** slots

Changing Parent Pointers



p jump.
p eatFly.
p parent: prince.
p dance.

Changing Parent Pointers



p jump.
p eatFly.
p parent: prince.
p dance.

Disadvantages of classes?

- Classes require programmers to understand a more complex model.
 - To make a new kind of object, we have to create a new class first.
 - To change an object, we have to change the class.
 - Infinite meta-class regression.
- **But:** Does Self require programmer to reinvent structure?
 - Common to structure Self programs with *traits*: objects that simply collect behavior for sharing.

Recall JavaScript Prototypes

- Every JavaScript object has a prototype
 - Object literals linked to `Object.prototype`
 - Otherwise, prototype based on constructor

```
function Foo() {  
    this.x = 1;  
}  
obj = new Foo;
```

- Changing the JavaScript prototype
 - The prototype property is immutable
 - Changes to prototype property inherited immediately

Outline

- Central concepts in object-oriented languages
 - Dynamic lookup, encapsulation, subtyping, inheritance
- Objects as activation records
 - Simula – implementation as activation records with static scope
- Pure dynamically-typed object-oriented languages
 - Object implementation and run-time lookup
 - Class-based languages (Smalltalk)
 - Prototype-based languages (Self, JavaScript)
- ➔ Statically-typed object-oriented languages (next lecture)
 - C++ – using static typing to eliminate search
 - problems with C++ multiple inheritance
 - Java – using Interfaces to avoid multiple inheritance

