

"A programming language is low level when its programs require attention to the irrelevant."
- Alan J. Perlis.

CSE341

Programming Languages

Lecture 10.1 – December 1, 2015

Object Oriented Programming

© 2013 Yakup Genç

Slides are taken from J. Mitchell

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
 - C++ – 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, synchronous or asynchronous 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:

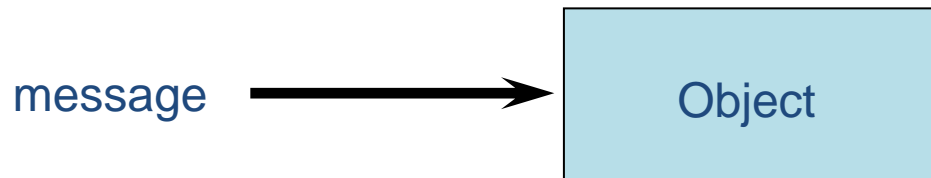
Overloading is resolved at compile time
Dynamic lookup is a run time operation

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 as **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 re-implementing functions that are already defined

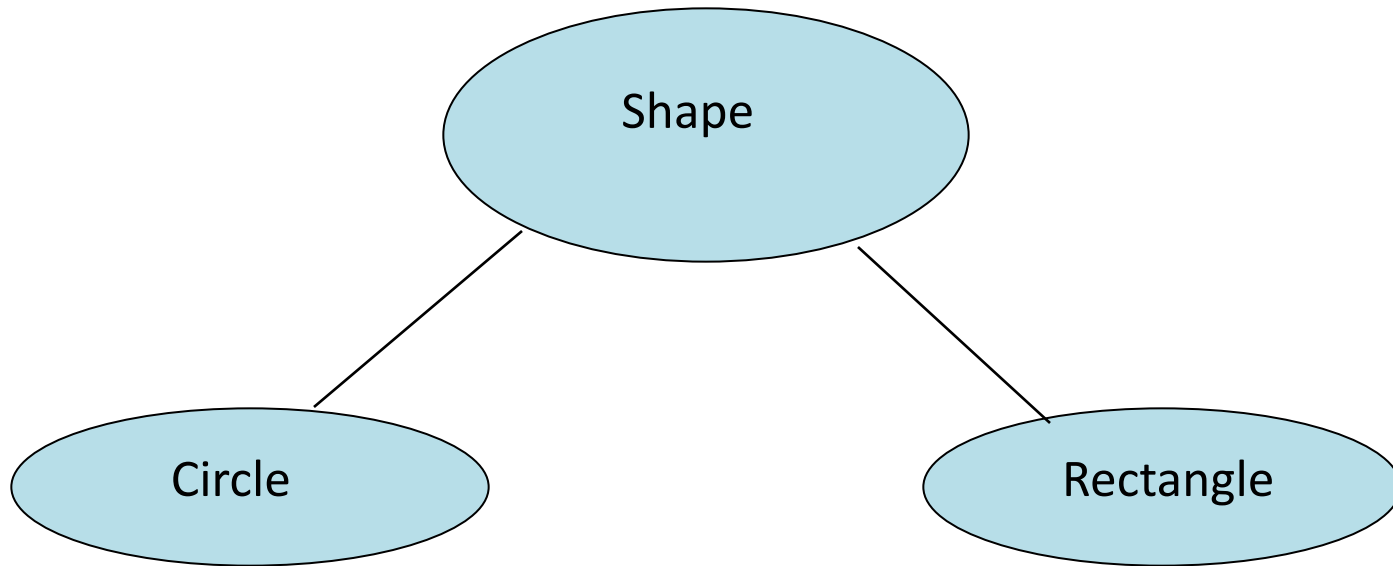
Example: Geometry Library

- 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
 - **Rectangle**: four points, representing corners
 - **Circle**: center point and radius

Subtype Hierarchy



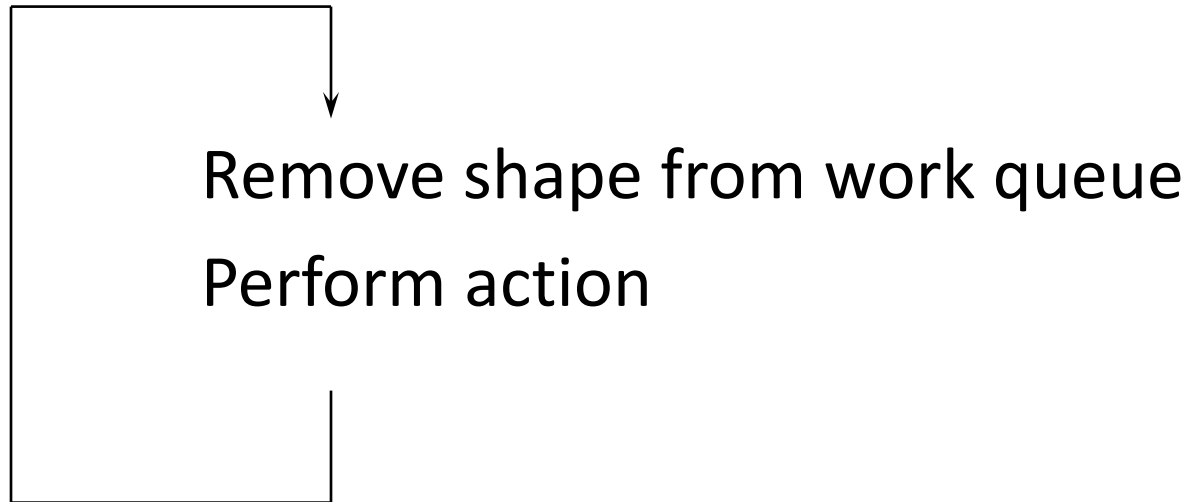
- General interface defined in the `shape` class
- Implementations defined in `circle`, `rectangle`
- Extend hierarchy with additional shapes

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
 - C++ – 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 programming 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

Outline

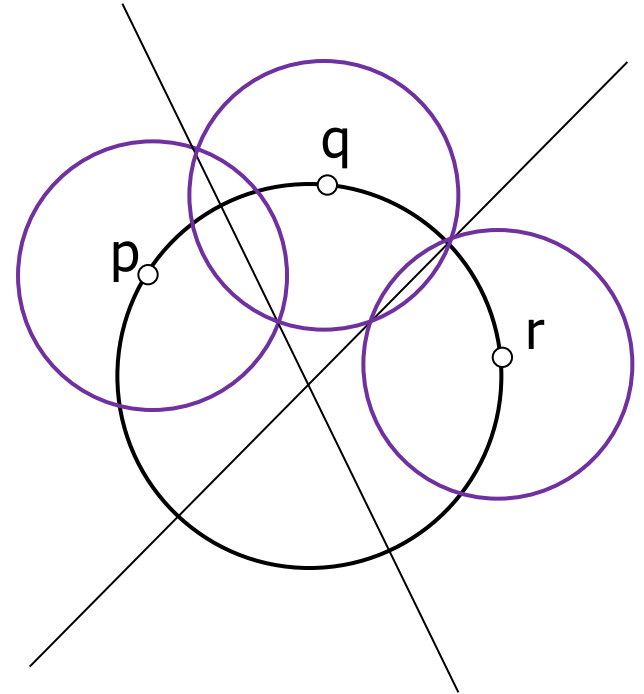
- Central concepts in object-oriented languages
 - Dynamic lookup, encapsulation, subtyping, inheritance
- Objects as activation records
 - Simula – implementation as activation records with static scope
- Pure dynamically-typed object-oriented languages
 - Object implementation and run-time lookup
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

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)

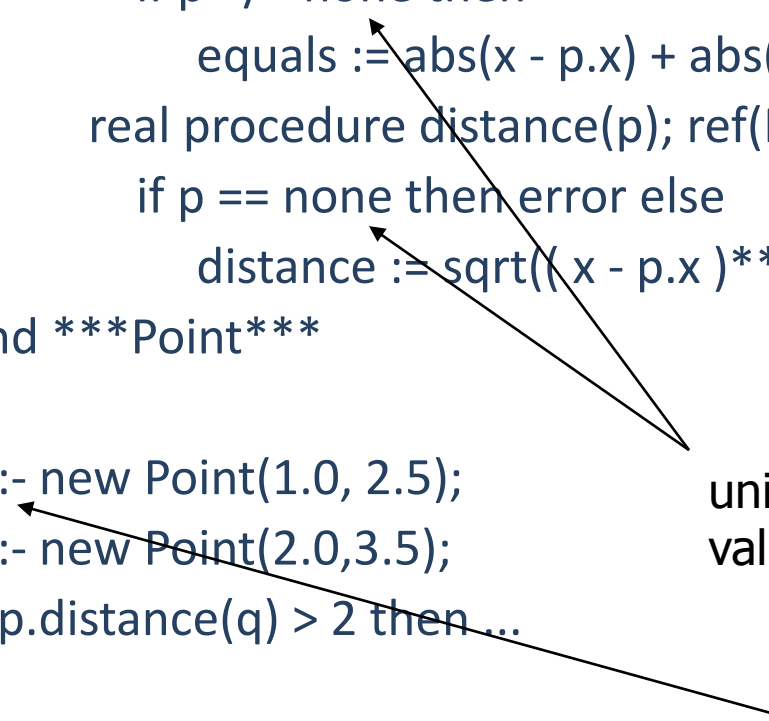
Simula Point Class

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

formal p is pointer to Point

uninitialized ptr has value none

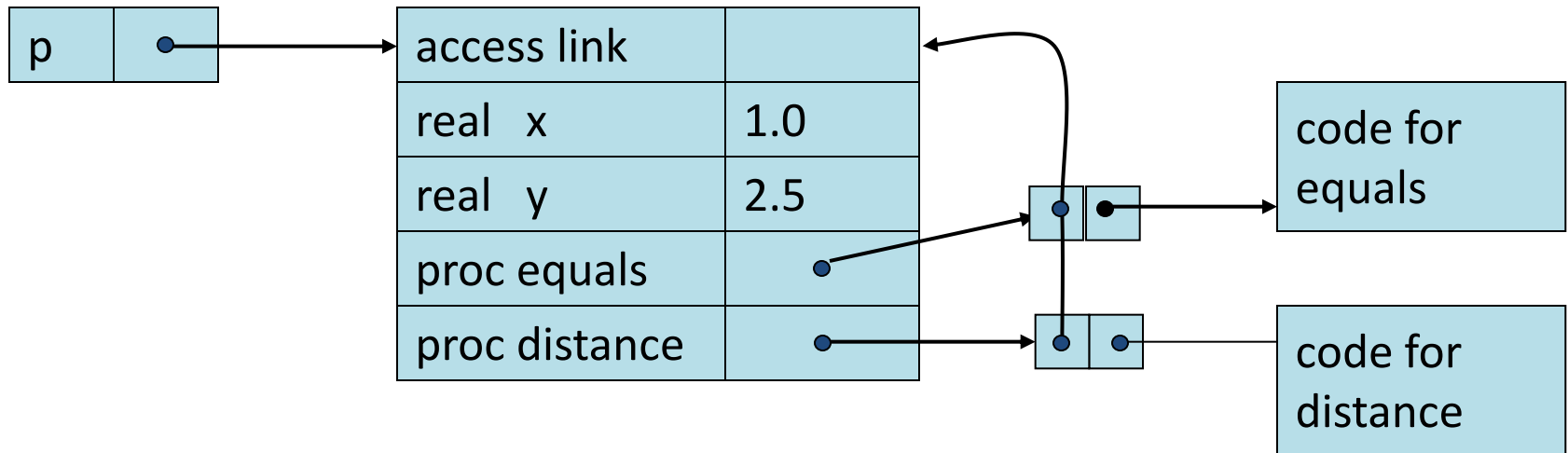
pointer assignment



Activation Records

- Modern imperative programming languages typically have local variables
 - Created upon entry to function
 - Destroyed when function returns
- Each invocation of a function has its own instantiation of local variables
 - Recursive calls to a function require several instantiations to exist simultaneously
 - Functions return only after all functions it calls have returned → last-in-first-out (LIFO) behavior.
 - A LIFO structure called a stack is used to hold each instantiation
- The portion of the stack used for an invocation of a function is called the function's activation record

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

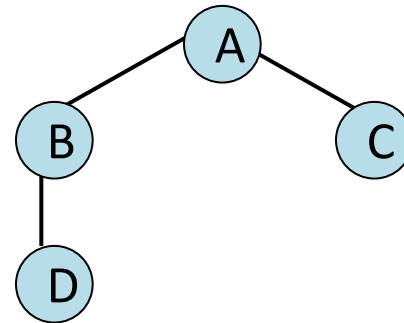
- A class declaration may be prefixed by a class name

class A

A class B

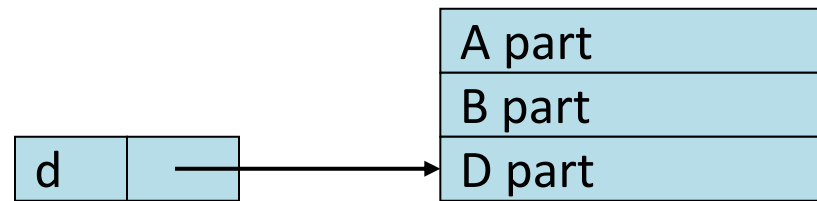
A class C

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

A closure is a function or reference to a function together with a referencing environment. A closure allows a function to access non-local variables even when invoked outside its immediate lexical scope.

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
 - C++ – 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 it is a very different language
 - 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...⟩	

Smalltalk Code

class name	Point	instance messages and methods
super class	Object	x: xcoord y: ycoord
instance var	x y	x <- xcoord
class var pi		y <- ycoord
		moveDx: dx Dy: dy
class messages and methods		x <- dx + x
	newX:xvalue Y:yvalue	y <- dy + y
	^ self new	x
	x: xvalue	^x
	y: yvalue	y
	newOrigin	^y
	^ self new	draw
	x: 0	
	y: 0	
	initialize	<< code to draw point >>
	pi <- 3.14159	

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 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...⟩

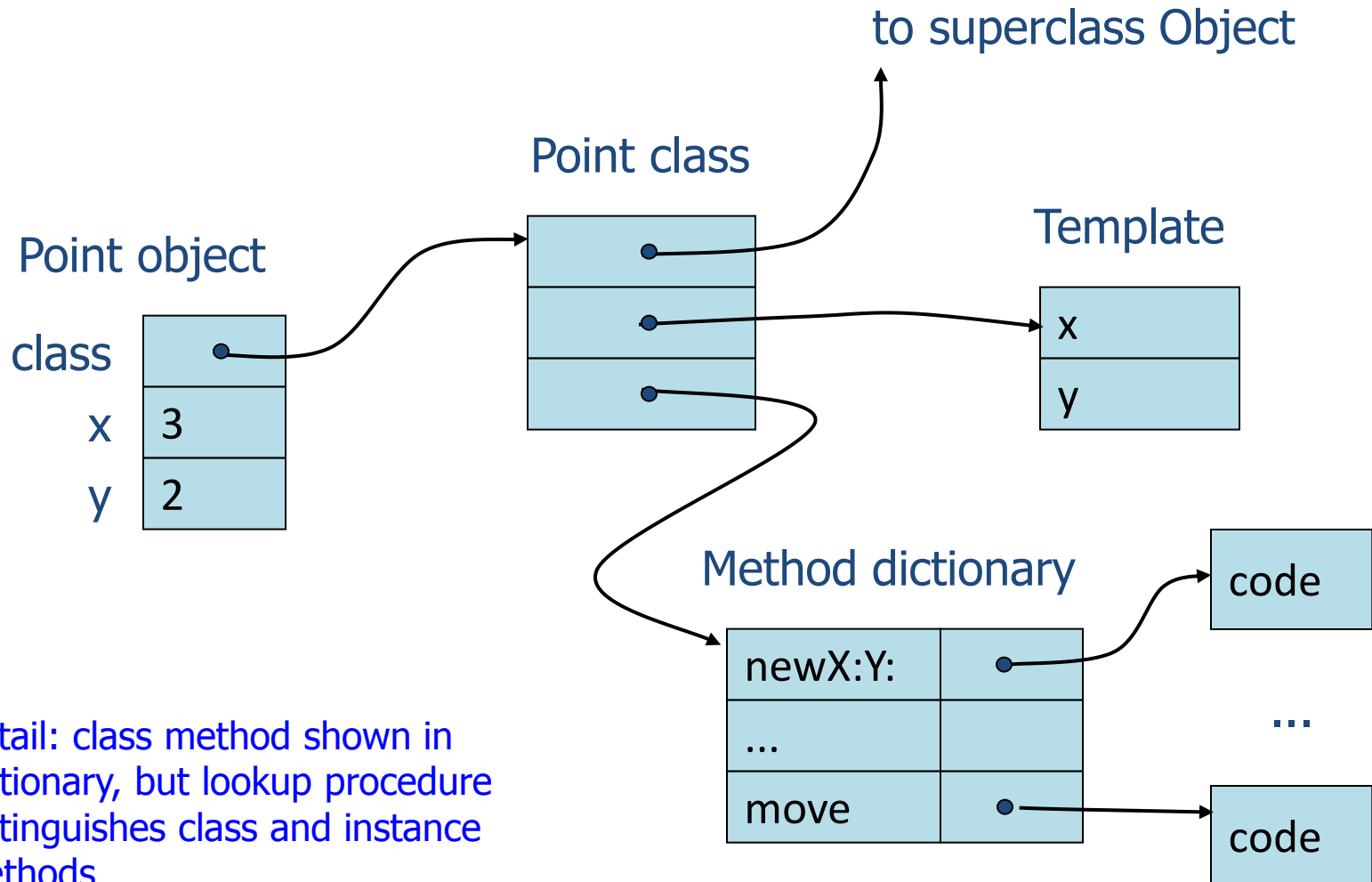
Explanation

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



Inheritance

- Define colored points from points

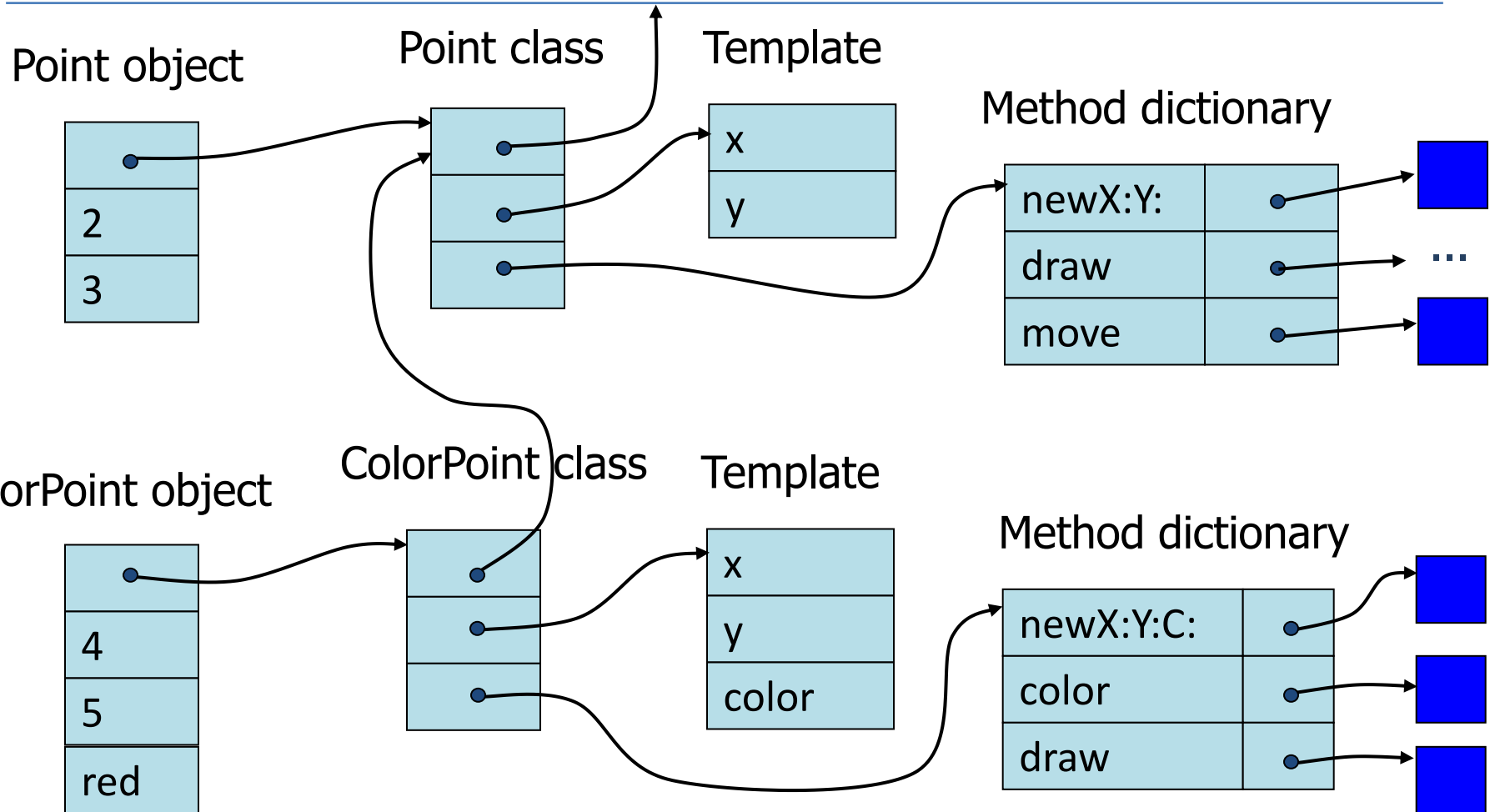
class name	ColorPoint
super class	Point
class var	
instance var	color
class messages and methods	
newX:xv Y:yv C:cv	< ... code ... >
instance messages and methods	
color	^color
draw	< ... code ... >

new instance
variable

new method

override Point
method

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

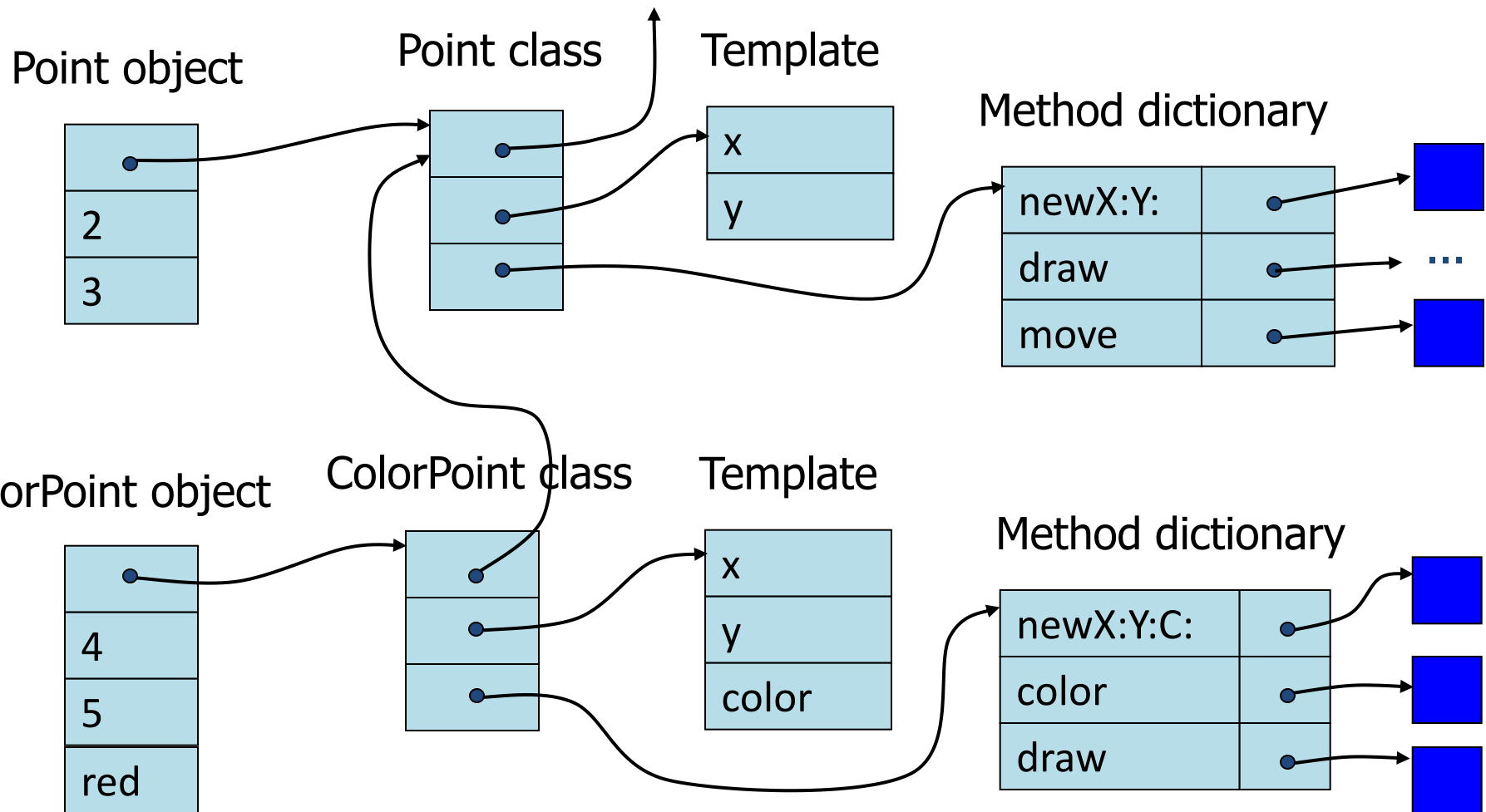
Self Programming Language

- 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 4.4 available from <http://selflanguage.org/>

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

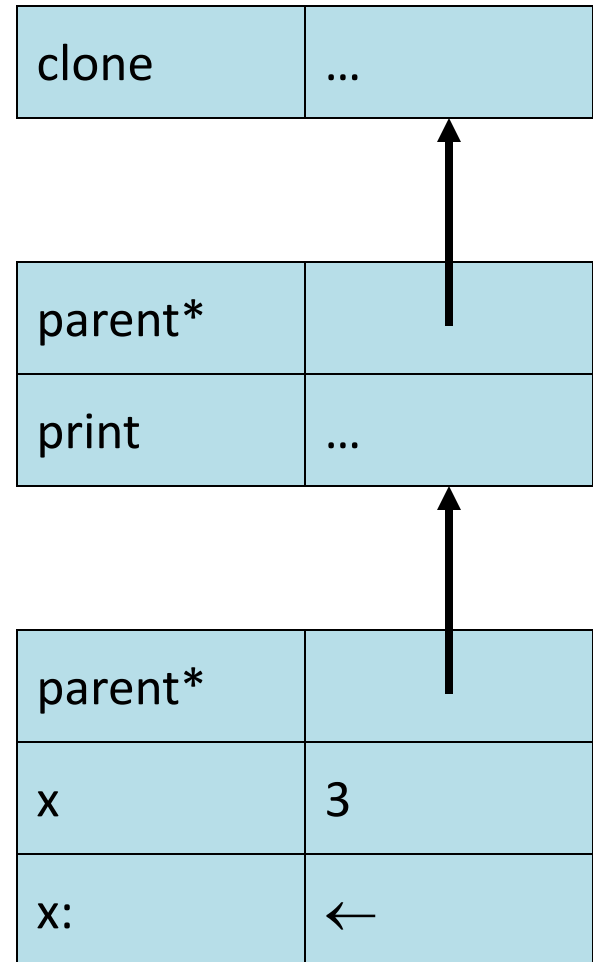
Objects and Slots

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

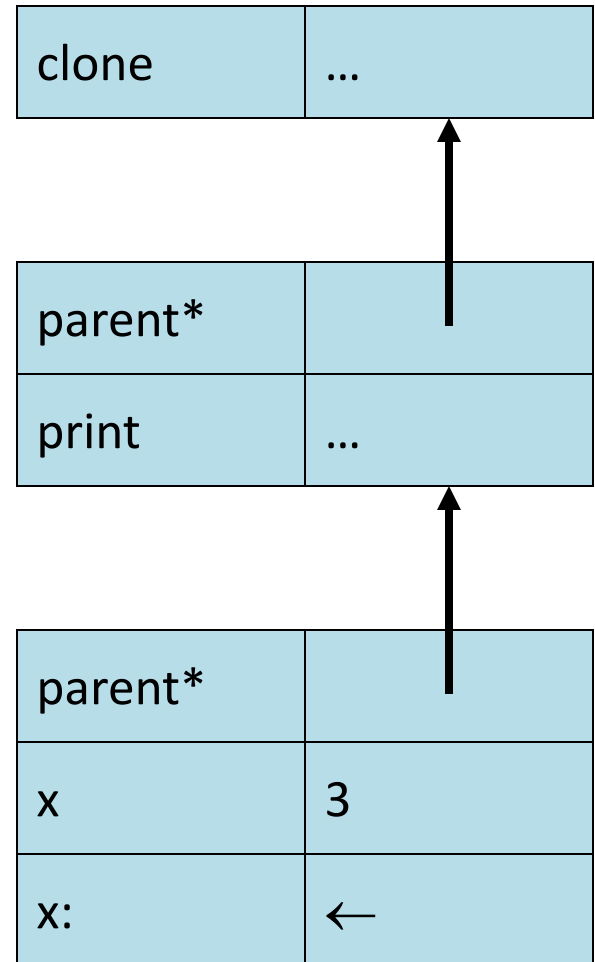
3

obj print

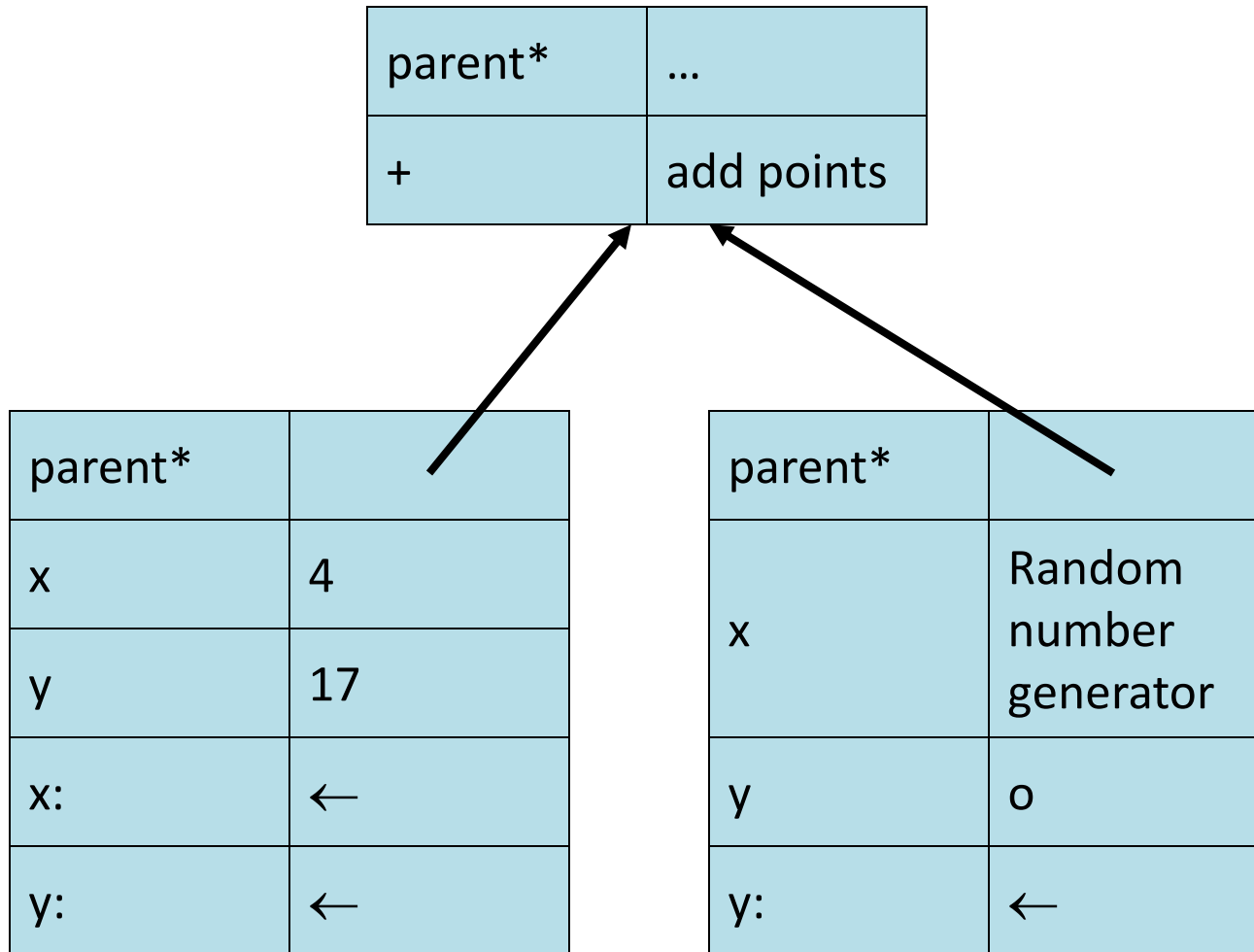
print point object

obj x: 4

obj after setting x to 4



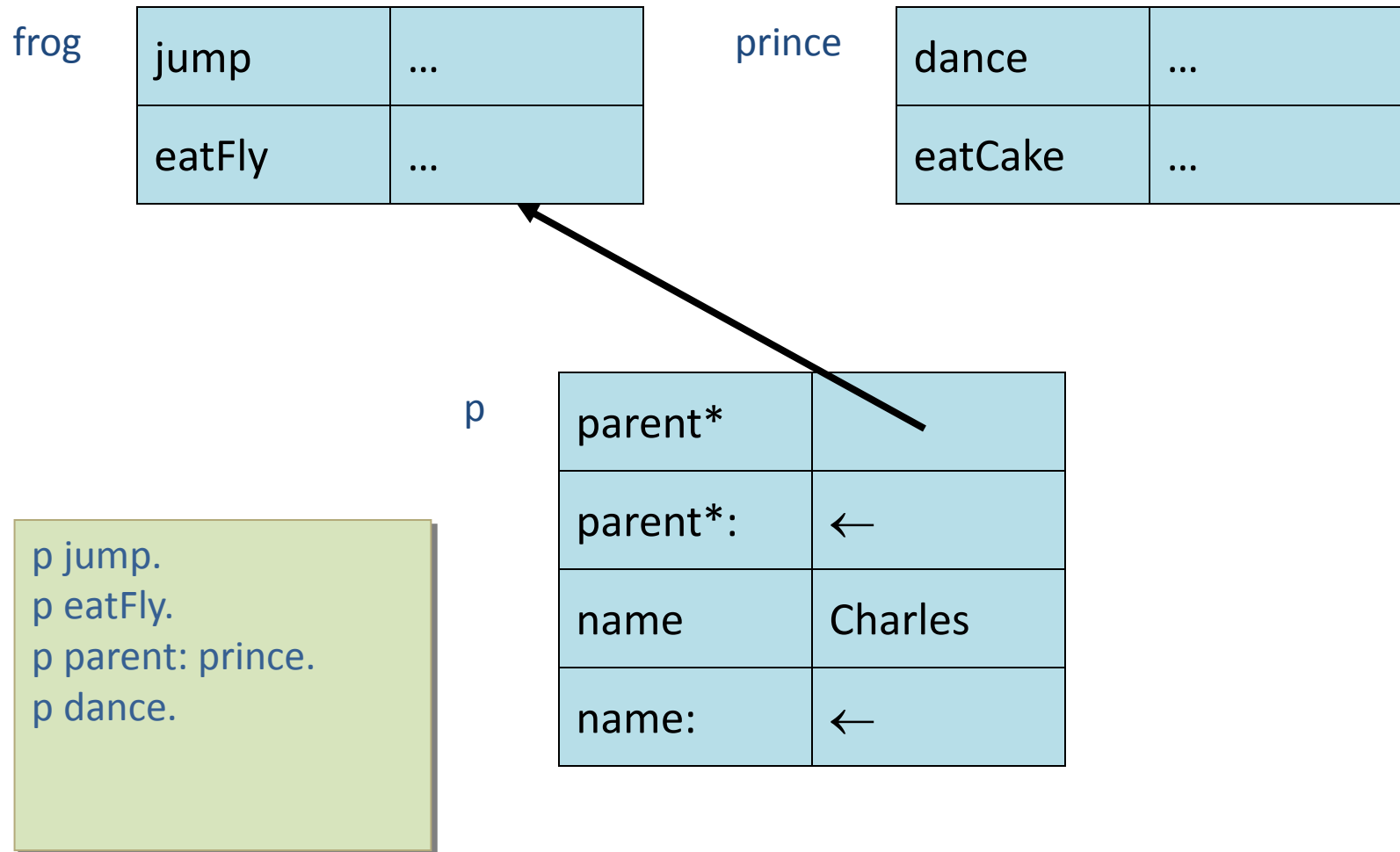
Mixing State and Behavior



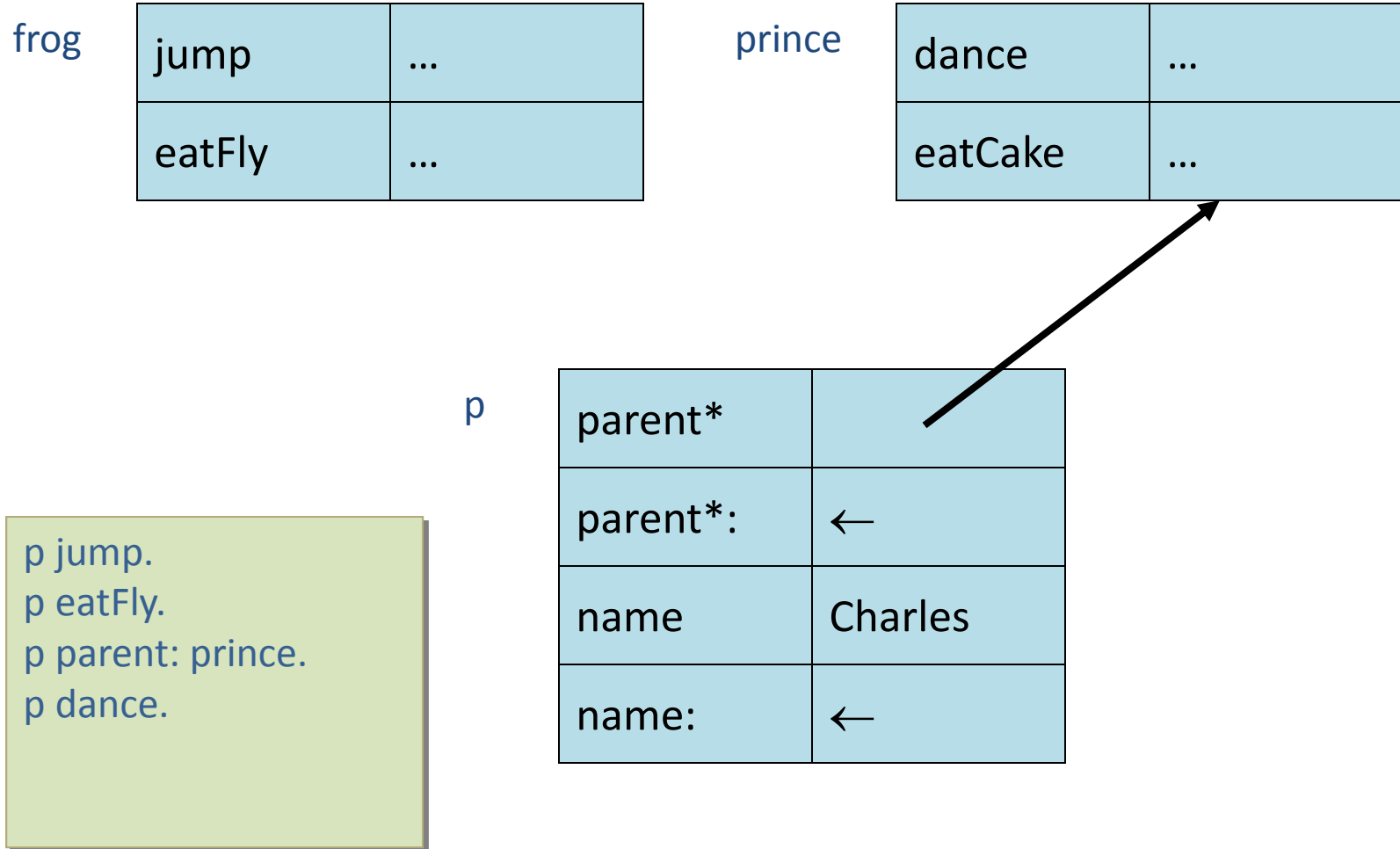
Object Creation

- 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



Changing Parent Pointers



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.

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
 - C++ – problems with C++ multiple inheritance
 - Java – using Interfaces to avoid multiple inheritance

C++ Background

- C++ is an object-oriented extension of C
- C was designed by Dennis Ritchie at Bell Labs
 - used to write Unix, based on BCPL
- C++ designed by Bjarne Stroustrup at Bell Labs
 - His original interest at Bell was research on simulation
 - Early extensions to C are based primarily on Simula
 - Called “C with classes” in early 1980’s
 - Popularity increased in late 1980’s and early 1990’s
 - Features were added incrementally
 - Classes, templates, exceptions, multiple inheritance, type tests...

C++ Design Goals

- Provide object-oriented features in C-based language, without compromising efficiency
 - Backwards compatibility with C
 - Better static type checking
 - Data abstraction
 - Objects and classes
 - Prefer efficiency of compiled code where possible
- Important principle
 - If you do not use a feature, your compiled code should be as efficient as if the language did not include the feature.
(compare to Smalltalk)

How successful?

- Given the design goals and constraints,
 - this is a very well-designed language
- Many users -- tremendous popular success
- However, very complicated design
 - Many features with complex interactions
 - Difficult to predict from basic principles
 - Most users chose a subset of language
 - Full language is complex and unpredictable
 - Many implementation-dependent properties

Significant Constraints

- C has specific machine model
 - Access to underlying architecture
- No garbage collection
 - Consistent with goal of efficiency
 - Need to manage object memory explicitly
- Local variables stored in activation records
 - Objects treated as generalization of structs
 - Objects may be allocated on stack and treated as L-values
 - Stack/heap difference is visible to programmer

C++ Object System

- Object-oriented features
 - Classes
 - Objects, with dynamic lookup of virtual functions
 - Inheritance
 - Single and multiple inheritance
 - Public and private base classes
 - Subtyping
 - Tied to inheritance mechanism
 - Encapsulation
 - Public, private, protected visibility

Some Good Decisions

- Public, private, protected levels of visibility
 - Public: visible everywhere
 - Protected: within class and subclass declarations
 - Private: visible only in class where declared
- Friend functions and classes
 - Careful attention to visibility and data abstraction
- Allow inheritance without subtyping
 - Better control of subtyping than without private base classes

Some Problem Areas

- Casts
 - Sometimes no-op, sometimes not (e.g., multiple inheritance)
- Lack of garbage collection
 - Memory management is error prone
 - Constructors, destructors are helpful // smart pointers?
- Objects allocated on stack
 - Better efficiency, interaction with exceptions
 - But assignment works badly, possible dangling ptrs
- Overloading
 - Too many code selection mechanisms?
- Multiple inheritance
 - Emphasis on efficiency leads to complicated behavior

Sample Class: 1D Points

```
class Pt {  
    public:  
        Pt(int xv);  
        Pt(Pt* pv);  
        int getX();  
        virtual void move(int dx);  
    protected:  
        void setX(int xv);  
    private:  
        int x;  
};
```

} Overloaded constructor

Public read access to private data

Virtual function

Protected write access L

Private data

Virtual Functions

- Member functions are either
 - Virtual, if explicitly declared or inherited as virtual
 - Non-virtual otherwise
- Virtual functions
 - Accessed by indirection through ptr in object
 - May be redefined in derived (sub) classes
- Non-virtual functions
 - Are called in the usual way. *Just ordinary functions.*
 - Cannot redefine in derived classes (except overloading)
- Pay overhead only if you use virtual functions

Sample Derived Class

```
class ColorPt: public Pt {  
    public:  
        ColorPt(int xv,int cv);  
        ColorPt(Pt* pv,int cv);  
        ColorPt(ColorPt* cp);  
        int getColor();  
        virtual void move(int dx);  
        virtual void darken(int tint);  
    protected:  
        void setColor(int cv);  
    private:  
        int color;  
};
```

Public base class gives supertype

Overloaded constructor

Non-virtual function

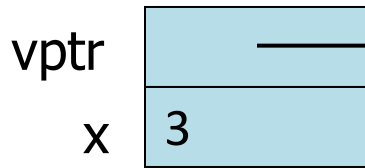
Virtual functions

Protected write access

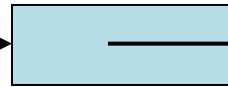
Private data

Run-time Representation

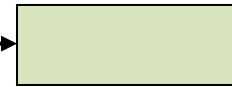
Point object



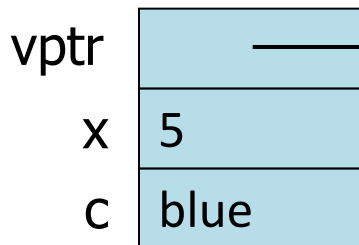
Point vtable



Code for move



ColorPoint object



ColorPoint vtable



Code for move



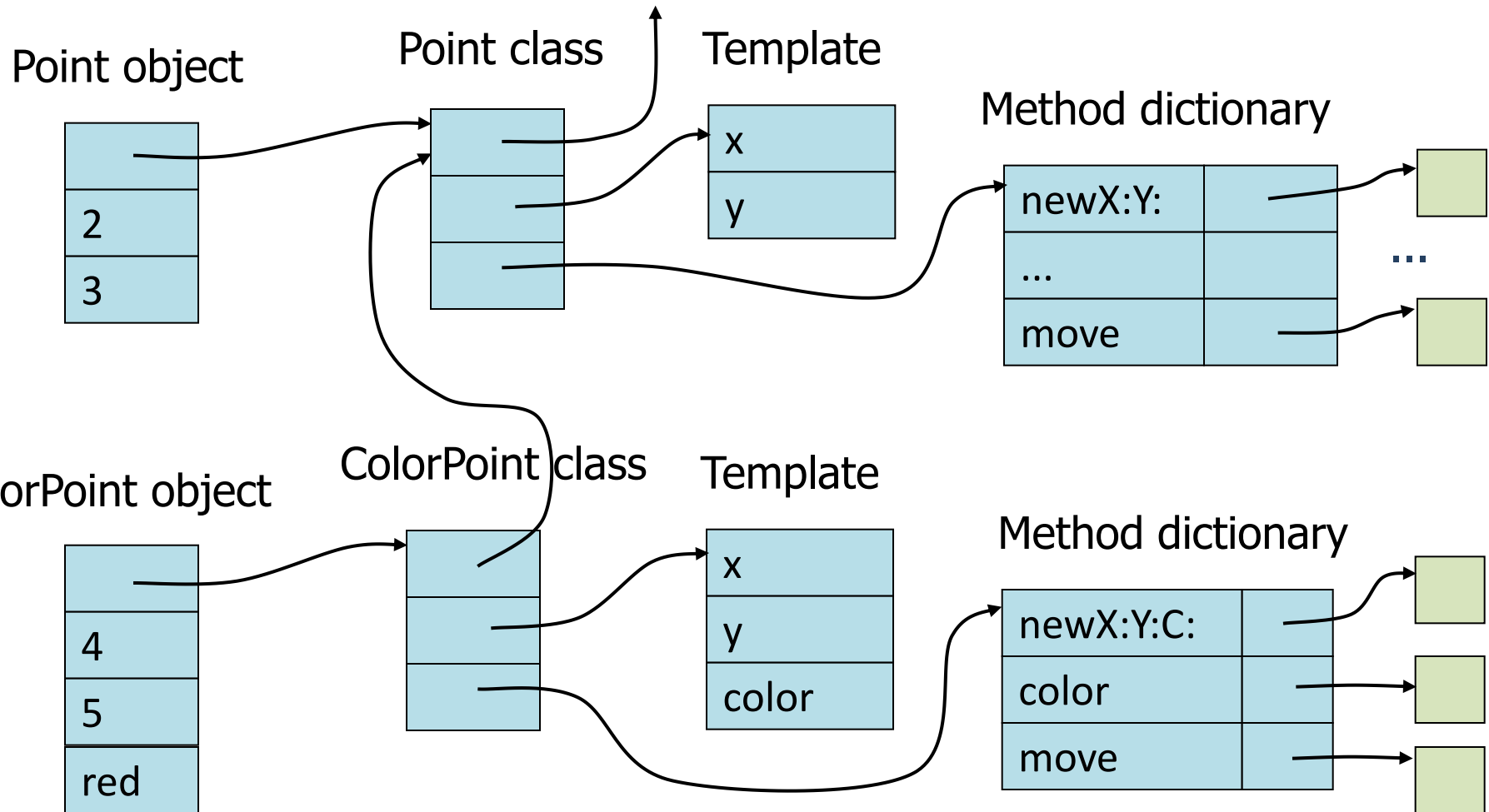
Code for darken



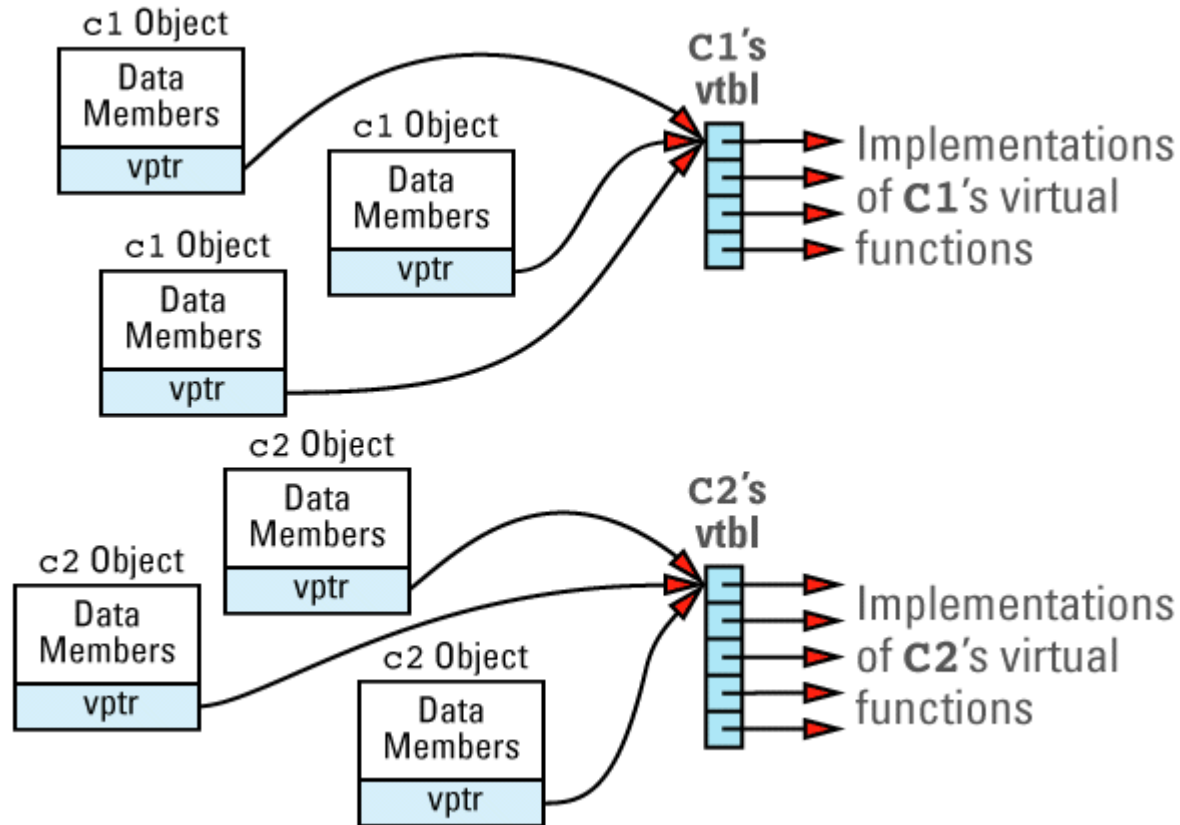
Data at same offset

Function pointers at same offset

Compare to Smalltalk/JavaScript



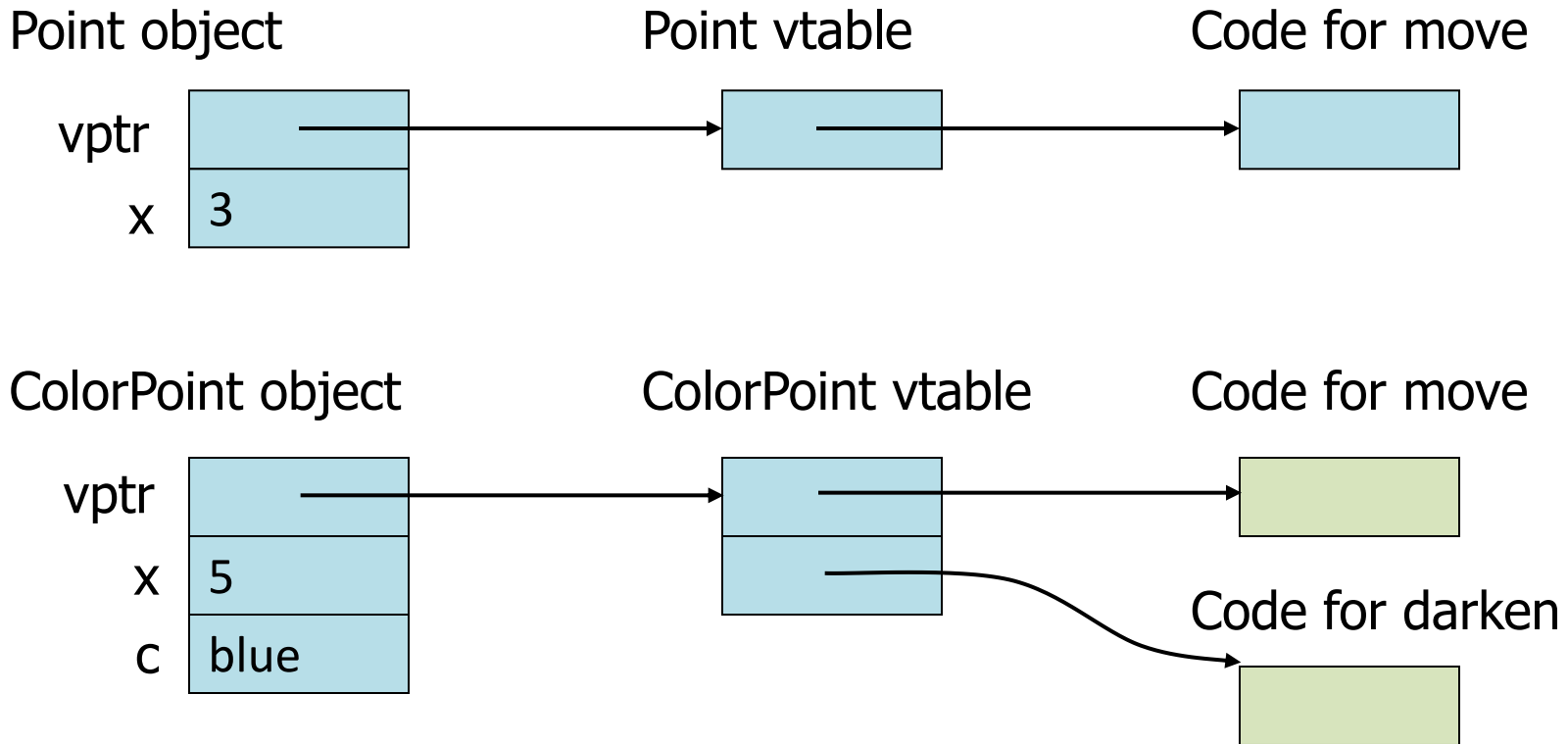
Run-time Representation



Why is C++ lookup simpler?

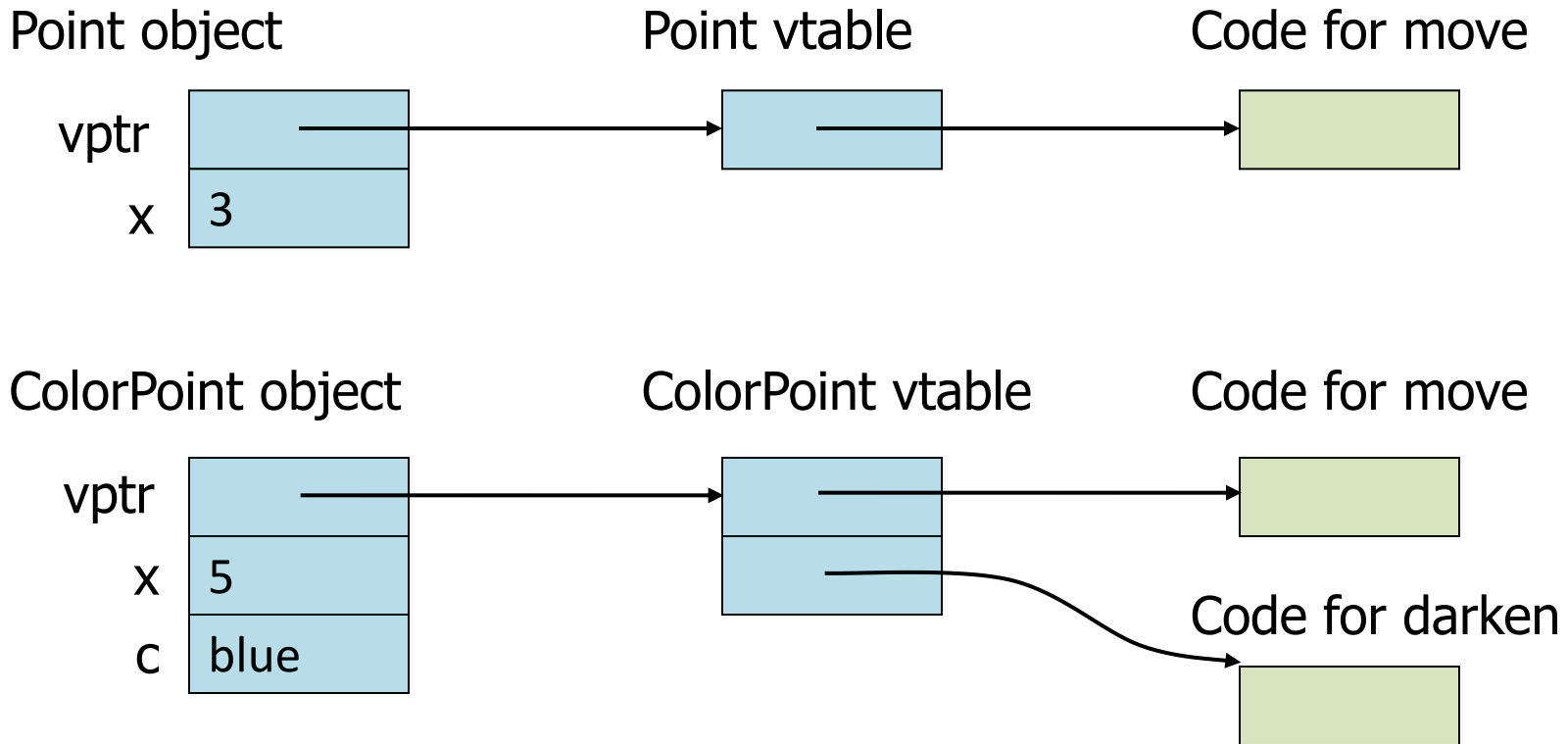
- Smalltalk/JavaScript have no static type system
 - Code `obj.operation(pars)` could refer to any object
 - Need to find method using pointer from object
 - Different classes will put methods at different place in method dictionary
- C++ type gives compiler some superclass
 - Offset of data, fctn ptr same in subclass and superclass
 - Offset of data and function ptr known at compile time
 - Code `p->move(x)` compiles to equivalent of `(* (p->vptr[0]))(p,x)` if `move` is `first` function in vtable

Looking Up Methods



```
Point p = new Pt(3);  
p->move(2);           // (*(p->vptr[0]))(p,2)
```


Looking Up Methods 2



```
Point cp = new ColorPt(5,blue);  
cp->move(2);    // (*(cp->vptr[0]))(cp,2)
```

Calls to Virtual Functions

- One member function may call another

```
class A {  
    public:  
        virtual int f(int x);  
        virtual int g(int y);  
};  
int A::f(int x) { ... g(i) ...;}  
int A::g(int y) { ... f(j) ...;}
```

- How does body of `f` call the right `g`?
 - If `g` is redefined in derived class `B`, then inherited `f` must call `B::g`

“This” Pointer (*self* in Smalltalk)

- Code is compiled so that member function takes “object itself” as first argument

Code `int A::f(int x) { ... g(i) ...;}`

compiled as `int A::f(A *this, int x) { ... this->g(i) ...;}`

- “this” pointer may be used in member function
 - Can be used to return pointer to object itself, pass pointer to object itself to another function, ...

Non-virtual Functions

- How is code for non-virtual function found?
- Same way as ordinary “non-member” functions:
 - Compiler generates function code and assigns address
 - Address of code is placed in symbol table
 - At call site, address is taken from symbol table and placed in compiled code
- Overloading
 - Remember: overloading is resolved at compile time
 - This is different from run-time lookup of virtual function

Virtual vs Overloaded Functions

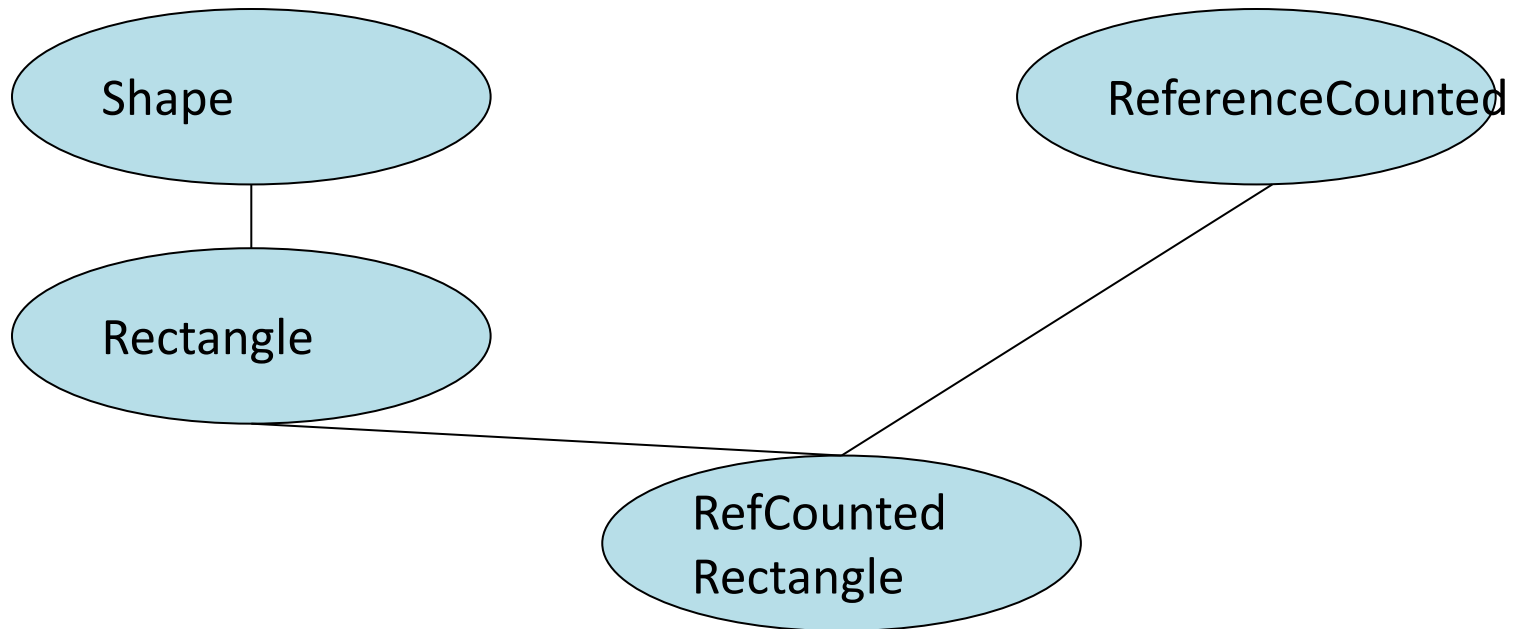
```
class parent { public:
    void printclass() {printf("p ");};
    virtual void printvirtual() {printf("p ");}; };
class child : public parent { public:
    void printclass() {printf("c ");};
    virtual void printvirtual() {printf("c ");}; };
main() {
    parent p; child c; parent *q;
    p.printclass(); p.printvirtual(); c.printclass(); c.printvirtual();
    q = &p; q->printclass(); q->printvirtual();
    q = &c; q->printclass(); q->printvirtual();
}
```

Output: p p c c p p ? ?

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
 - C++ – problems with C++ multiple inheritance
 - Java – using Interfaces to avoid multiple inheritance

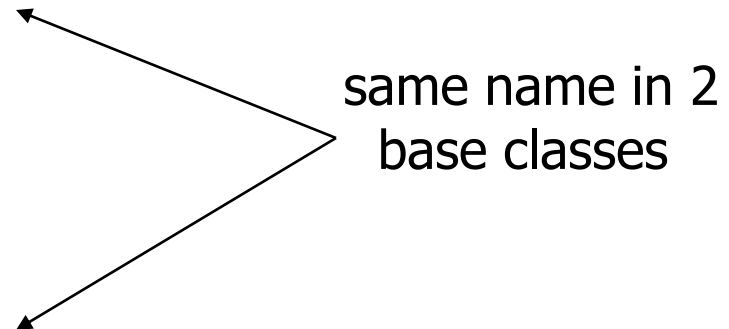
Multiple Inheritance



Inherit independent functionality from independent classes

Problem: Name Clashes

```
class A {  
    public:  
        void virtual f() { ... }  
};  
class B {  
    public:  
        void virtual f() { ... }  
};  
class C : public A, public B { ... };  
...  
    C* p;  
    p->f();    // error
```



Possible Solutions to Name Clash

- Three general approaches
 - Implicit resolution
 - Language resolves name conflicts with arbitrary rule
 - Explicit resolution
 - Programmer must explicitly resolve name conflicts
 - Disallow name clashes
 - Programs are not allowed to contain name clashes
- No solution is always best
- C++ uses explicit resolution

Repair to Previous Example

- Rewrite class `C` to call `A::f` explicitly

```
class C : public A, public B {  
    public:  
        void virtual f( ) {  
            A::f( );    // Call A::f(), not B::f();  
        }  
}
```

- Reasonable solution
 - This eliminates ambiguity
 - Preserves dependence on `A`
 - Changes to `A::f` will change `C::f`

vtable for Multiple Inheritance

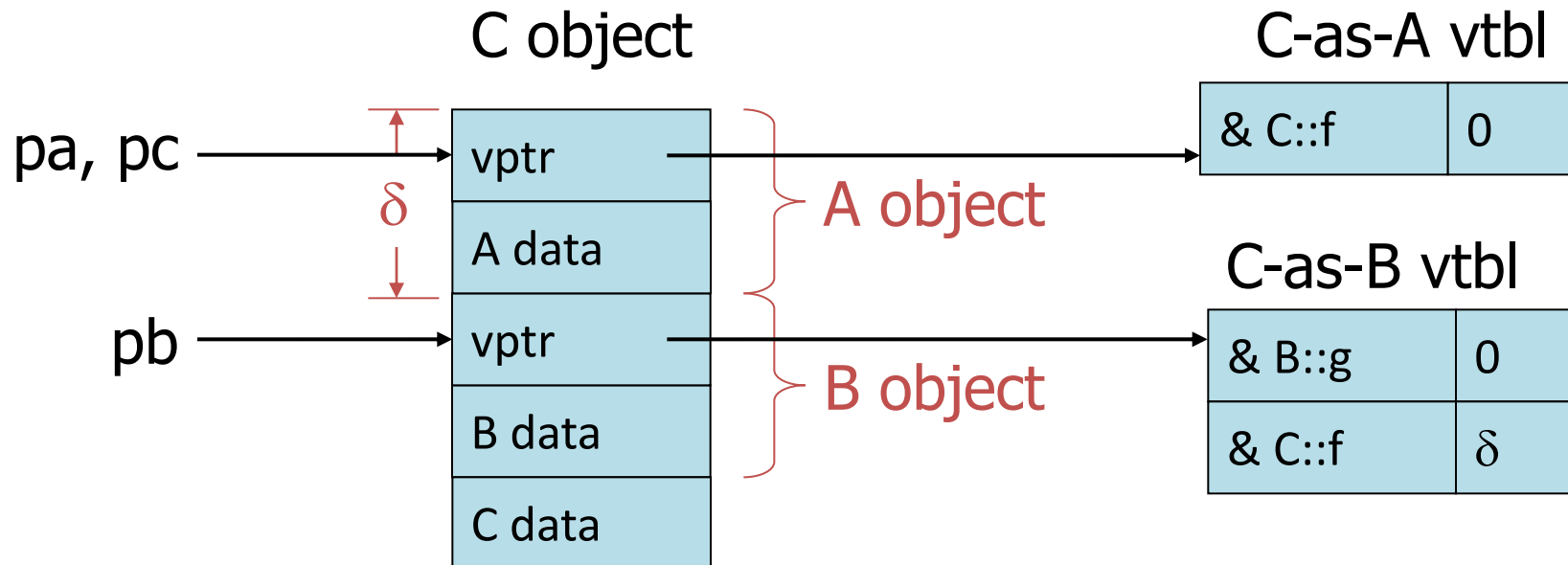
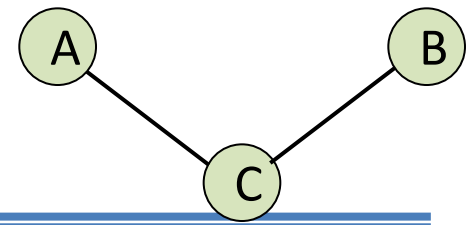
```
class A {  
    public:  
        int x;  
        virtual void f();  
};  
  
class B {  
    public:  
        int y;  
        virtual void g();  
        virtual void f();  
};
```

```
class C: public A, public B {  
    public:  
        int z;  
        virtual void f();  
};
```

```
C *pc = new C;  
B *pb = pc;  
A *pa = pc;
```

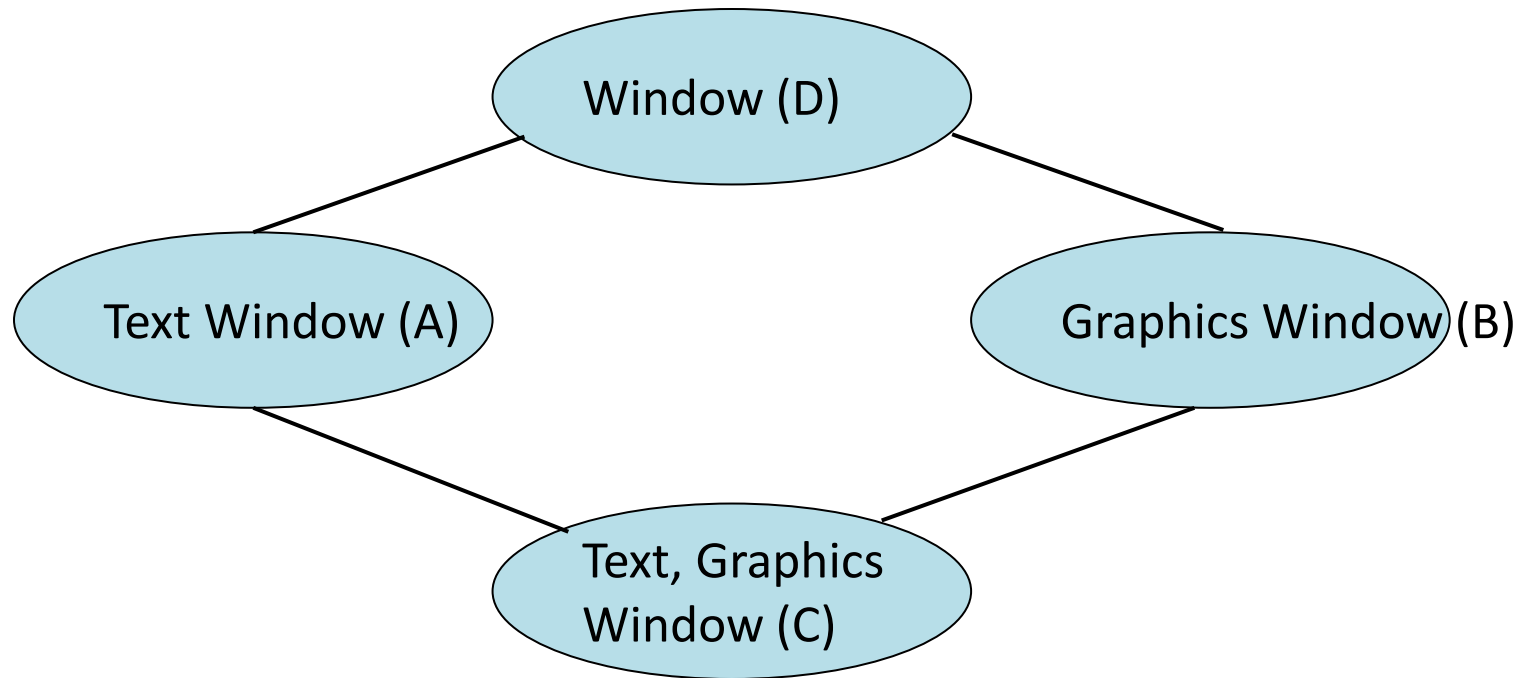
Three pointers to same object, but different static types.

Object and Classes



- Offset δ in vtbl is used in call to $pb \rightarrow f$, since $C::f$ may refer to A data that is above the pointer pb
- Call to $pc \rightarrow g$ can proceed through $C\text{-as-}B$ vtbl

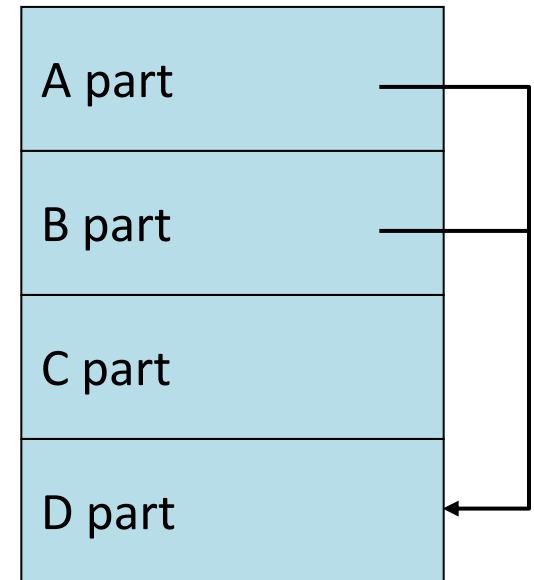
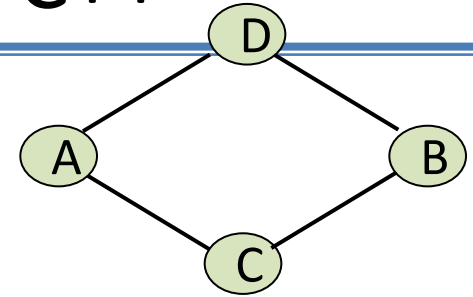
Multiple Inheritance “Diamond”



- Is interface or implementation inherited twice?
- What if definitions conflict?

Diamond Inheritance in C++

- Standard base classes
 - D members appear twice in C
- Virtual base classes
 - `class A : public virtual D { ... }`
 - Avoid duplication of base class members
 - Require additional pointers so that D part of A, B parts of object can be shared



C++ multiple inheritance is complicated
because of desire to maintain efficient lookup

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
 - C++ – problems with C++ multiple inheritance
 - Java – using Interfaces to avoid multiple inheritance

Java Language Background

- James Gosling and others at Sun, 1990 - 95
- Oak language for “set-top box”
 - small networked device with television display
 - graphics
 - execution of simple programs
 - communication between local program and remote site
 - no “expert programmer” to deal with crash, etc.
- Internet applications
 - simple language for writing programs that can be transmitted over network
 - not an integrated web scripting language like JavaScript

Design Goals

- Portability
 - Internet-wide distribution: PC, Unix, Mac
- Reliability
 - Avoid program crashes and error messages
- Safety
 - Programmer may be malicious
- Simplicity and familiarity
 - Appeal to average programmer; less complex than C++
- Efficiency
 - Important but secondary

General Design Decisions

- Simplicity
 - Almost everything is an object
 - All objects on heap, accessed through pointers
 - No functions, no multiple inheritance, no go to, no operator overloading, few automatic coercions
- Portability and network transfer
 - Bytecode interpreter on many platforms
- Reliability and Safety
 - Typed source and typed bytecode language
 - Run-time type and bounds checks
 - Garbage collection

Language Terminology

- Class, object – as in other languages
- Field – data member
- Method – member function
- Static members – class fields and methods
- this – self
- Package – set of classes in shared namespace
- Native method – method compiled from in another language, often C

Java Classes and Objects

- Syntax similar to C++
- Object
 - has fields and methods
 - is allocated on heap, not run-time stack
 - accessible through reference (only ptr assignment)
 - garbage collected
- Dynamic lookup
 - Similar in behavior to other languages
 - Static typing => more efficient than Smalltalk
 - Dynamic linking, interfaces => slower than C++

Point Class

```
class Point {  
    private int x;  
    protected void setX (int y) {x = y;}  
    public int  getX()    {return x;}  
    Point(int xval) {x = xval;}    // constructor  
};
```

- Visibility similar to C++, but not exactly (later slide)

Object Initialization

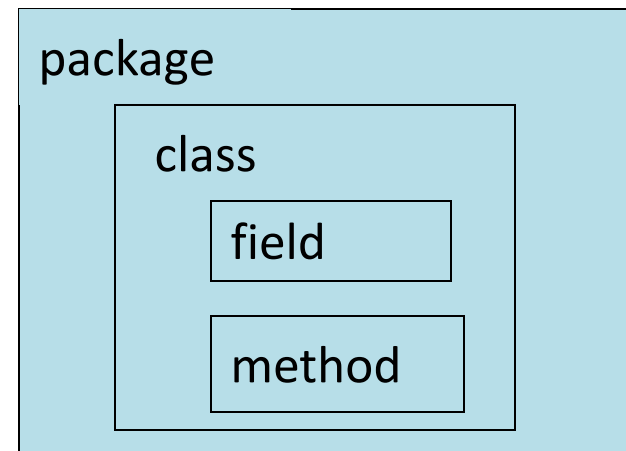
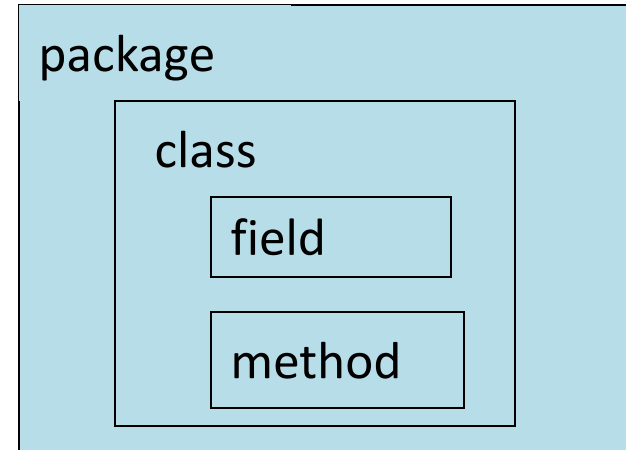
- Java guarantees constructor call for each object
 - Memory allocated
 - Constructor called to initialize memory
 - Some interesting issues related to inheritance
- Cannot do this (would be bad C++ style anyway):
 - `Obj* obj = (Obj*)malloc(sizeof(Obj));`
- Static fields of class initialized at class load time

Garbage Collection and Finalize

- Objects are garbage collected
 - No explicit *free*
 - Avoids dangling pointers and resulting type errors
- Problem
 - What if object has opened file or holds lock?
- Solution
 - *finalize* method, called by the garbage collector
 - Before space is reclaimed, or when virtual machine exits
 - Space overflow is not really the right condition to trigger finalization when an object holds a lock...)
 - Important convention: call `super.finalize`

Encapsulation and Packages


- Every field, method belongs to a class
- Every class is part of some package
 - Can be unnamed default package
 - File declares which package code belongs to



Visibility and Access

- Four visibility distinctions
 - public, private, protected, package
- Method can refer to
 - private members of class it belongs to
 - non-private members of all classes in same package
 - protected members of superclasses (in diff package)
 - public members of classes in visible packages

Visibility determined by files system, etc. (outside language)
- Qualified names (or use import)
 - `java.lang.String.substring()`


package class method

Inheritance

- Similar to Smalltalk, C++
- Subclass inherits from superclass
 - Single inheritance only (but Java has interfaces)
- Some additional features
 - Conventions regarding *super* in constructor and *finalize* methods
 - Final classes and methods

Example Subclass

```
class ColorPoint extends Point {  
    // Additional fields and methods  
    private Color c;  
    protected void setC (Color d) {c = d;}  
    public Color  getC()   {return c;}  
    // Define constructor  
    ColorPoint(int xval, Color cval) {  
        super(xval); // call Point constructor  
        c = cval; }   // initialize ColorPoint field  
};
```

Class *Object*

- Every class extends another class
 - Superclass is *Object* if no other class named
- Methods of class *Object*
 - getClass – return the Class object representing class of the object
 - toString – returns string representation of object
 - equals – default object equality (not ptr equality)
 - hashCode
 - Clone – makes a duplicate of an object
 - wait, notify, notifyAll – used with concurrency
 - finalize

Constructors and Super

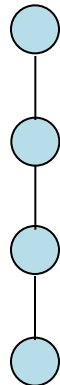
- Java guarantees constructor call for each object
- This must be preserved by inheritance
 - Subclass constructor must call super constructor
 - If first statement is not call to super, then call super() inserted automatically by compiler
 - If superclass does not have a constructor with no args, then this causes compiler error
 - Exception to rule: if one constructor invokes another, then it is responsibility of second constructor to call super, e.g.,

```
ColorPoint() { ColorPoint(0,blue);}
```

is compiled without inserting call to super
- Different conventions for finalize and super
 - Compiler does not force call to super finalize

Final Classes and Methods

- Restrict inheritance
 - Final classes and methods cannot be redefined
- Example
 - `java.lang.String`
- Reasons for this feature
 - Important for security
 - Programmer controls behavior of all subclasses
 - Critical because subclasses produce subtypes
 - Compare to C++ virtual/non-virtual
 - Method is “virtual” until it becomes final



Java Interfaces (by example)

```
interface Shape {  
    public float center();  
    public void rotate(float degrees);  
}  
  
interface Drawable {  
    public void setColor(Color c);  
    public void draw();  
}  
  
class Circle implements Shape, Drawable {  
    // does not inherit any implementation  
    // but must define Shape, Drawable methods  
}
```

Interfaces vs Multiple Inheritance

- C++ multiple inheritance
 - A single class may inherit from two base classes
 - Constraints of C++ require derived class representation to resemble *all* base classes
- Java interfaces
 - A single class may implement two interfaces
 - No inheritance (of implementation) involved
 - Java implementation does not require similarity between class representations
 - For now, think of Java implementation as Smalltalk/JavaScript implementation, although the Java type system supports some optimizations

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

Example: Smalltalk Point class

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...〉	

Subclass: ColorPoint

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

add instance variable

add method

override Point method

Object Interfaces

- Interface

The messages understood by an object

- Example: point

`x:y`: set x,y coordinates of point

`moveDx:Dy`: method for changing location

`x` returns x-coordinate of a point

`y` returns y-coordinate of a point

`draw` display point in x,y location on screen

- 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:y:
moveDx:Dy:
x
y
draw

Colored_point

x:y:
moveDx:Dy:
x
y
color
draw

Colored_point interface contains Point
Colored_point is a subtype of Point

Implicit Object Types – Smalltalk/JS

- Each object has an interface
 - Smalltalk: set of instance methods declared in class
 - Example:
`Point` { x:y:, moveDx:Dy:, x, y, draw}
`ColorPoint` { x:y:, moveDx:Dy:, x, y, color, draw}
 - This is a form of type
Names of methods, does not include type/protocol of arguments
- Object expression and type
 - Send message to object
`p draw` `p x:3 y:4`
`q color` `q moveDx: 5 Dy: 2`
 - Expression OK if message is in interface

Subtyping

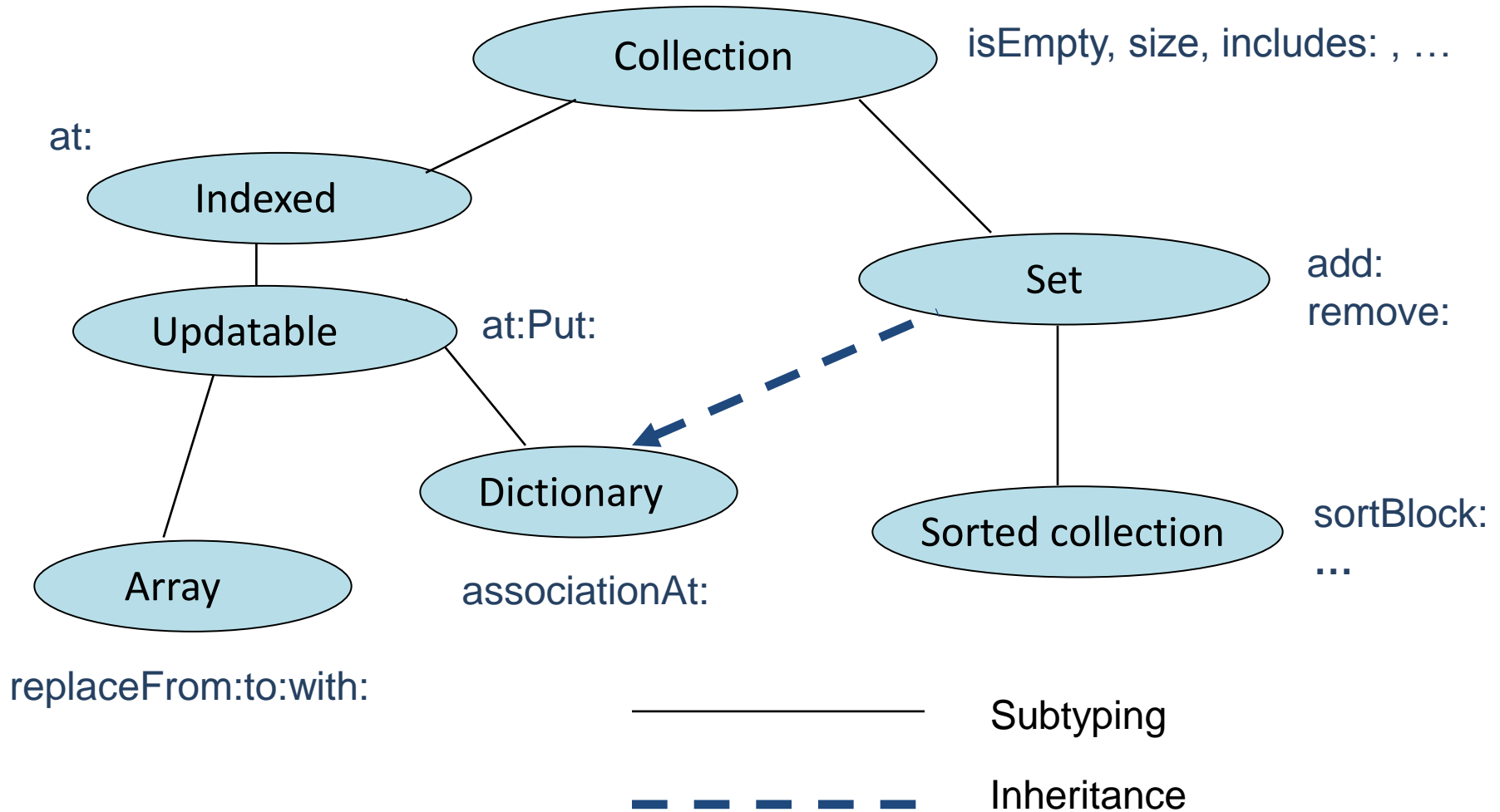
- Relation between interfaces
 - Suppose expression makes sense
p msg:pars -- OK if msg is in interface of p
 - Replace p by q if interface of q contains interface of p
- Subtyping
 - If interface is superset, then a subtype
 - Example: ColorPoint subtype of Point
 - Sometimes called “conformance”

Can extend to more detailed interfaces that include types of parameters

Subtyping and Inheritance

- Smalltalk/JavaScript subtyping is implicit
 - Not a part of the programming language
 - Important aspect of how systems are built
- Inheritance is explicit
 - Used to implement systems
 - No forced relationship to subtyping

Smalltalk Collection Hierarchy



C++ Subtyping

- Subtyping in principle
 - $A <: B$ if every A object can be used without type error whenever a B object is required
 - Example:

Point:	int getX();	}	Public members
	void move(int);		
ColorPoint:	int getX();	}	Public members
	int getColor();		
	void move(int);		
	void darken(int tint);		

- C++: $A <: B$ if class A has public base class B

Implementation of Subtyping

- No-op
 - Dynamically-typed languages
 - C++ object representations (single-inheritance only)

```
circle *c = new Circle(p,r);
```

```
shape *s = c;           // s points to circle c
```

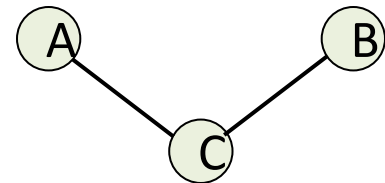
- Conversion
 - C++ object representations w/multiple-inheritance

```
C *pc = new C;
```

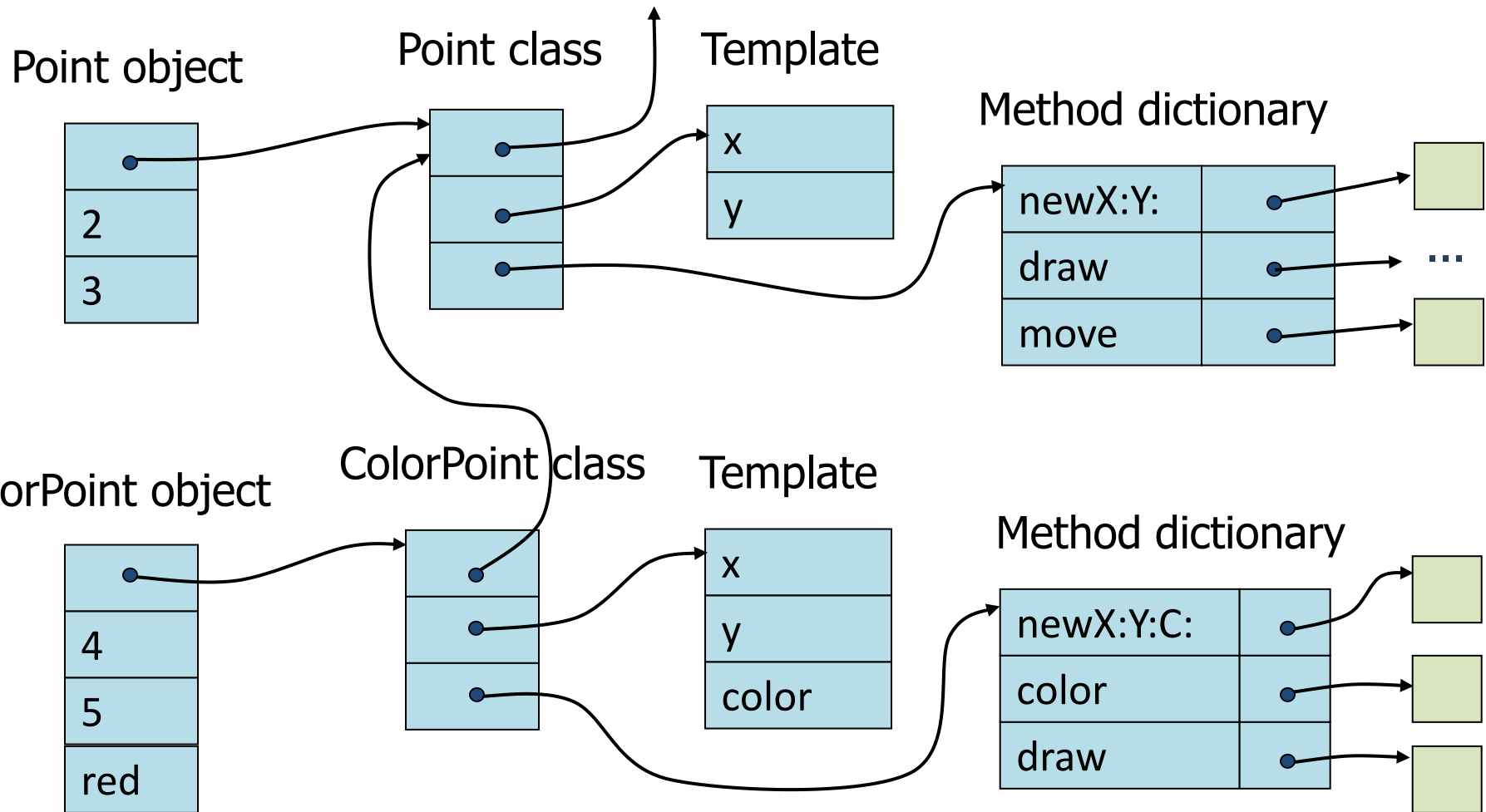
```
B *pb = pc;
```

```
A *pa = pc;
```

```
// may point to different position in object
```

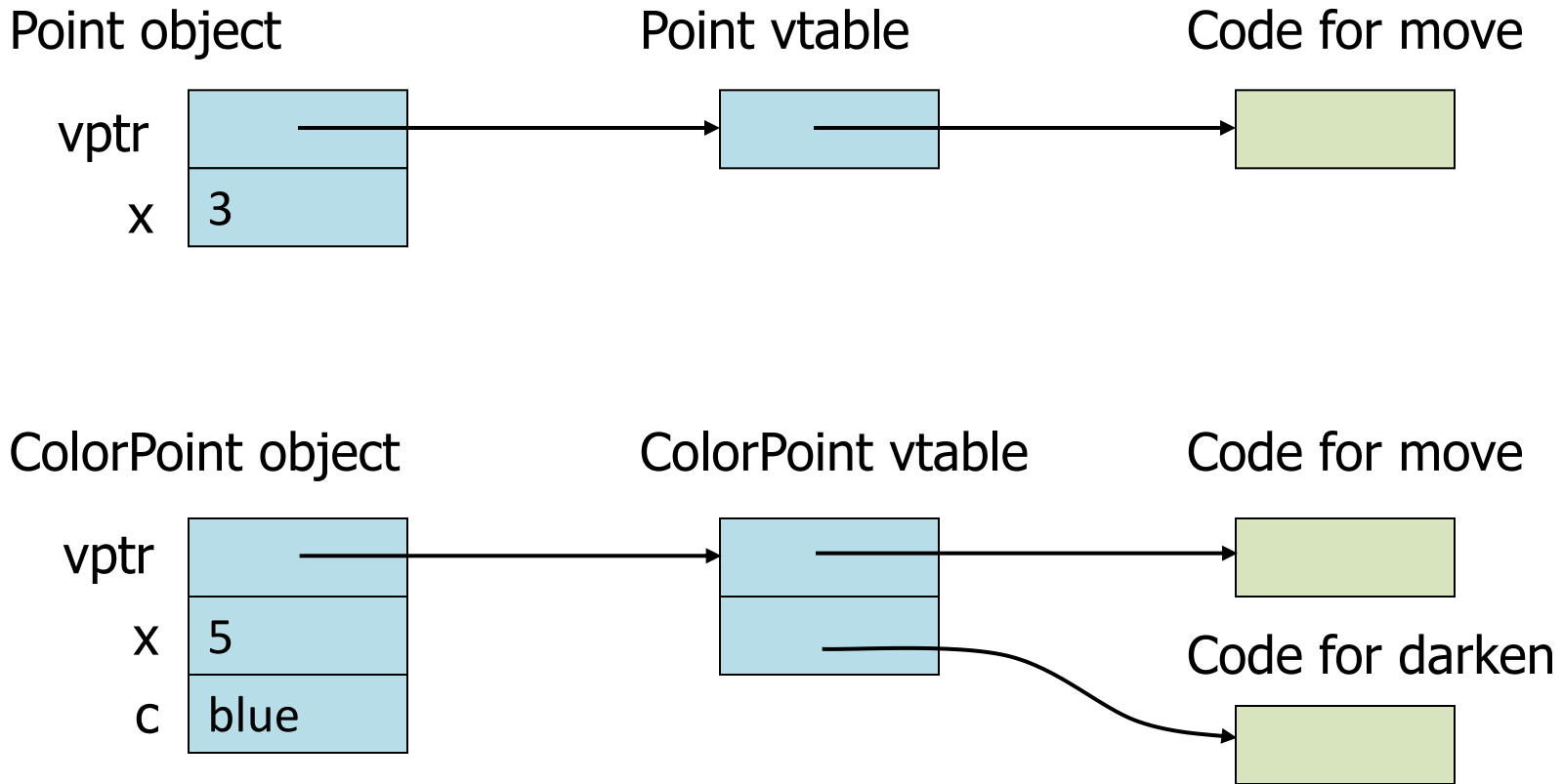


Smalltalk/JavaScript Representation



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

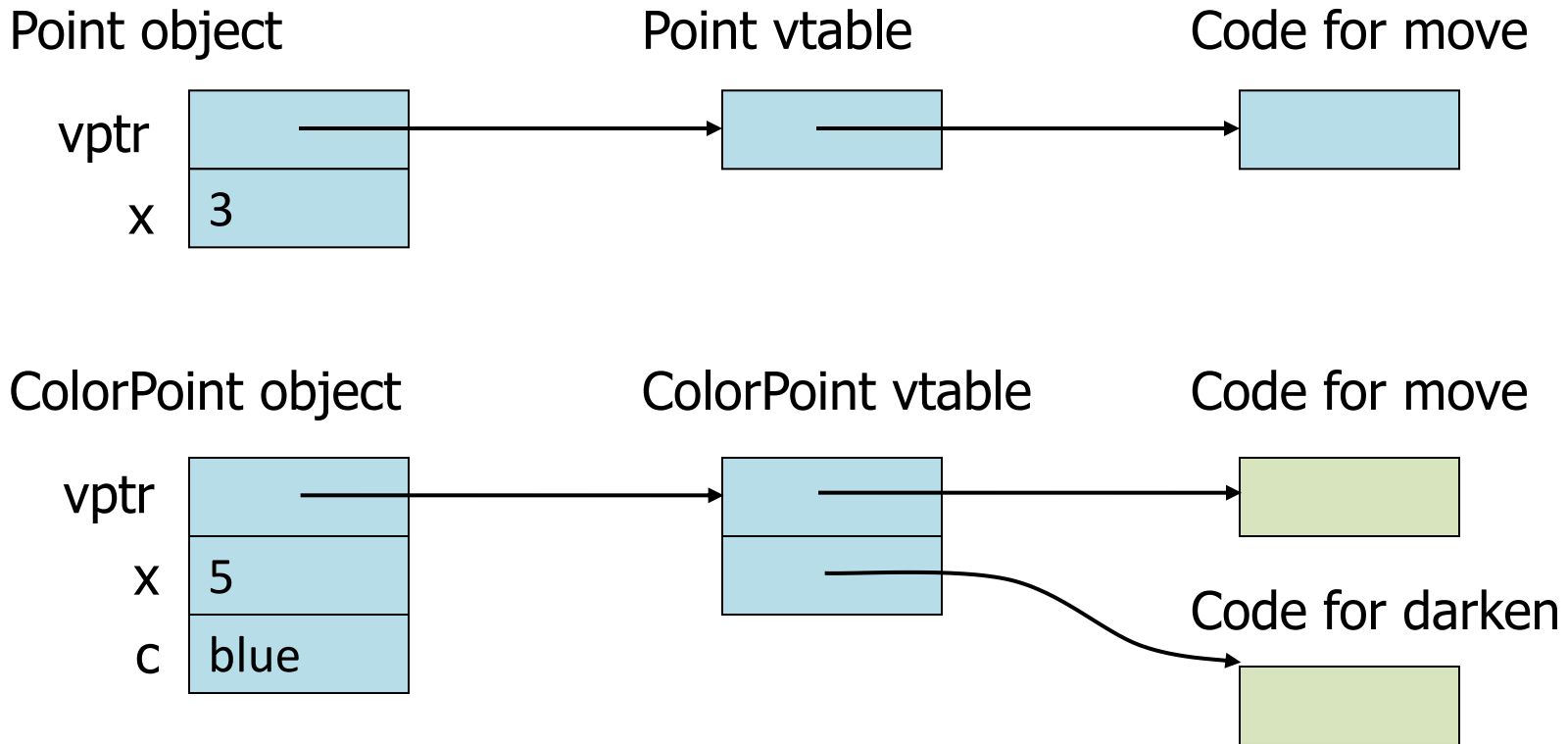
C++ Run-time Representation



Data at same offset

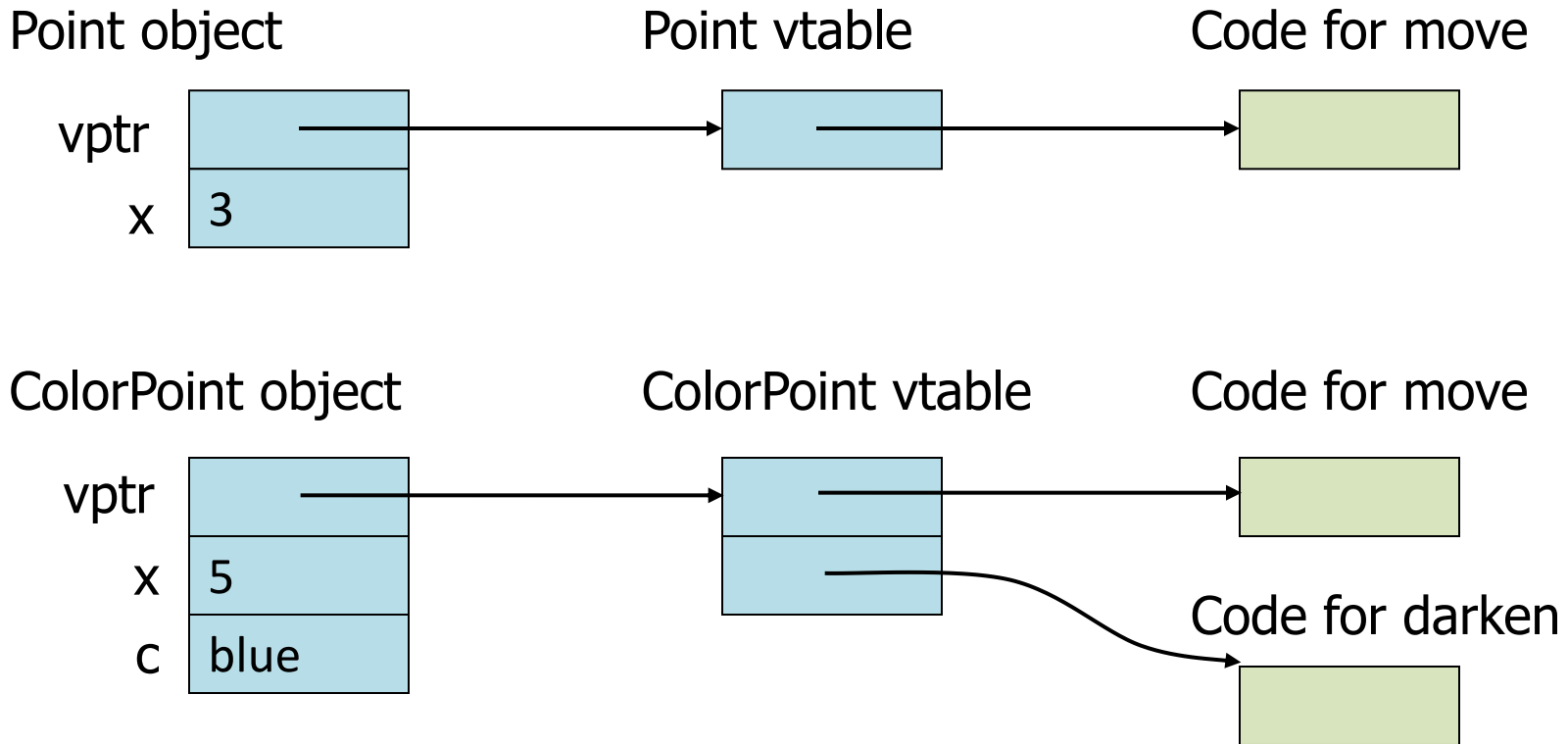
Function pointers at same offset

C++: Virtual Function Lookup



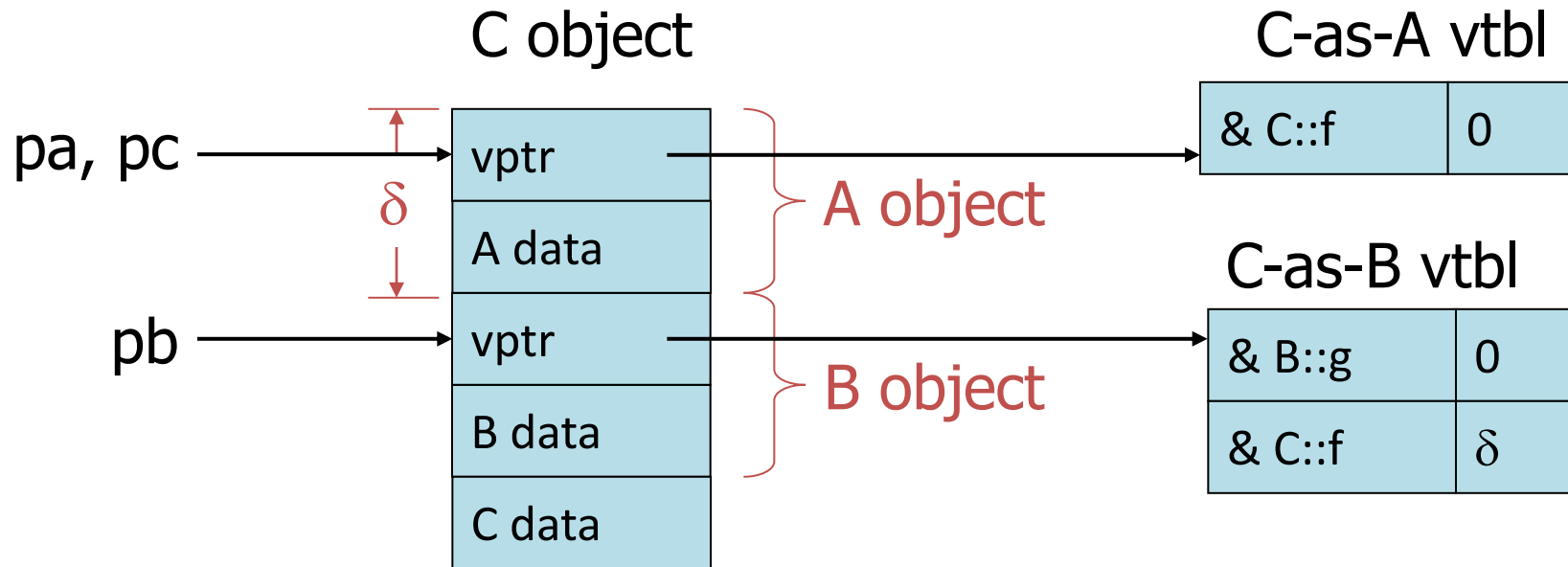
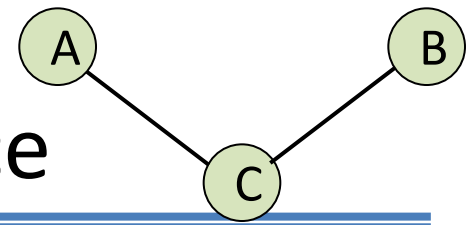
```
Point p = new Pt(3);  
p->move(2);           // (*(p->vp_ptr[0]))(p,2)
```

C++: Virtual Function Lookup 2



```
Point cp = new ColorPt(5,blue);
cp->move(2);    // (*(cp->vptr[0]))(cp,2)
```

C++ Multiple Inheritance



- Offset δ in vtbl is used in call to `pb->f`, since `C::f` may refer to `A` data that is above the pointer `pb`
- Call to `pc->g` can proceed through C-as-B vtbl

Independent Classes not Subtypes

```
class Point {  
    public:  
        int getX();  
        void move(int);  
    protected:    ...  
    private:      ...  
};
```

```
class ColorPoint {  
    public:  
        int getX();  
        void move(int);  
        int getColor();  
        void darken(int);  
    protected:    ...  
    private:      ...  
};
```

C++ does not treat `ColorPoint <: Point` as written

- **Need public inheritance** `ColorPoint : public Point`
- Recall: All public and protected members of base become private members of derived. However, derived can re-declare all to be public or protected (except originally protected).

Why C++ design?

- Client code depends only on public interface
 - In principle, if ColorPoint interface contains Point interface, then any client could use ColorPoint in place of point
 - However -- offset in virtual function table may differ
 - Lose implementation efficiency (like Smalltalk)
- Without link to inheritance
 - Subtyping leads to loss of implementation efficiency
- Also encapsulation issue:
 - Subtyping based on inheritance is preserved under modifications to base class ...

Recurring Subtype Issue: Downcast

- The Simula type of an object is its class
- Simula downcasts are checked at run-time
- Example:

```
class A(...); ...
```

```
A class B(...); ...
```

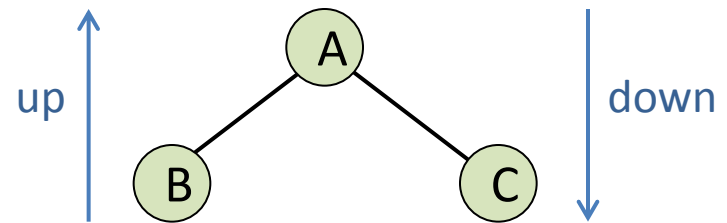
```
ref (A) a :- new A(...)
```

```
ref (B) b :- new B(...)
```

```
a := b      /* OK since B is subclass of A */
```

```
...
```

```
b := a      /* compiles, but run-time test */
```

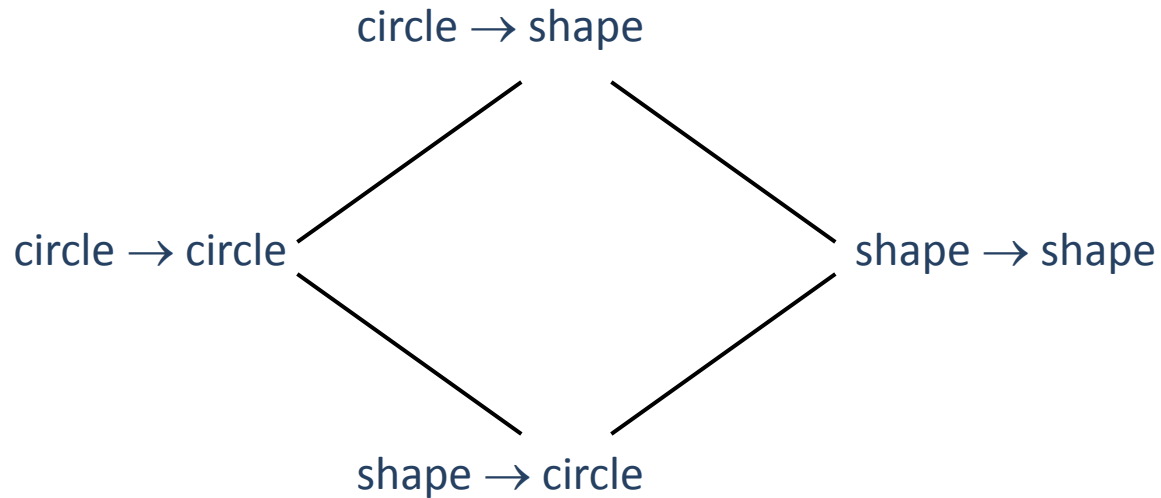


Function Subtyping

- Subtyping principle
 - $A <: B$ if an A expression can be safely used in any context where a B expression is required
- Subtyping for function results
 - If $A <: B$, then $C \rightarrow A <: C \rightarrow B$
- Subtyping for function arguments
 - If $A <: B$, then $B \rightarrow C <: A \rightarrow C$
- Terminology
 - Covariance: $A <: B$ implies $F(A) <: F(B)$
 - Contravariance: $A <: B$ implies $F(B) <: F(A)$

Examples

- If `circle <: shape`, then



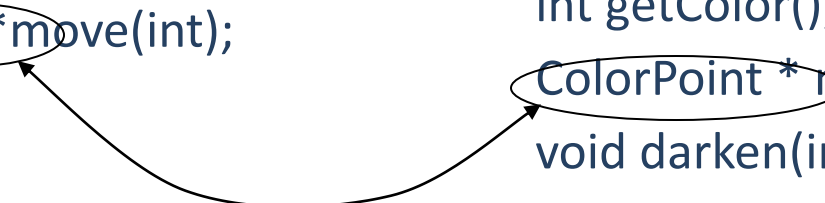
C++ compilers recognize limited forms of function subtyping

Subtyping with Functions

```
class Point {  
  public:  
    int getX();  
    virtual Point *move(int);  
  protected: ...  
  private: ...  
};
```

```
class ColorPoint: public Point {  
  public:  
    int getX();  
    int getColor();  
    ColorPoint *move(int);  
    void darken(int);  
  protected: ...  
  private: ...  
};
```

↙ Inherited, but repeated here for clarity



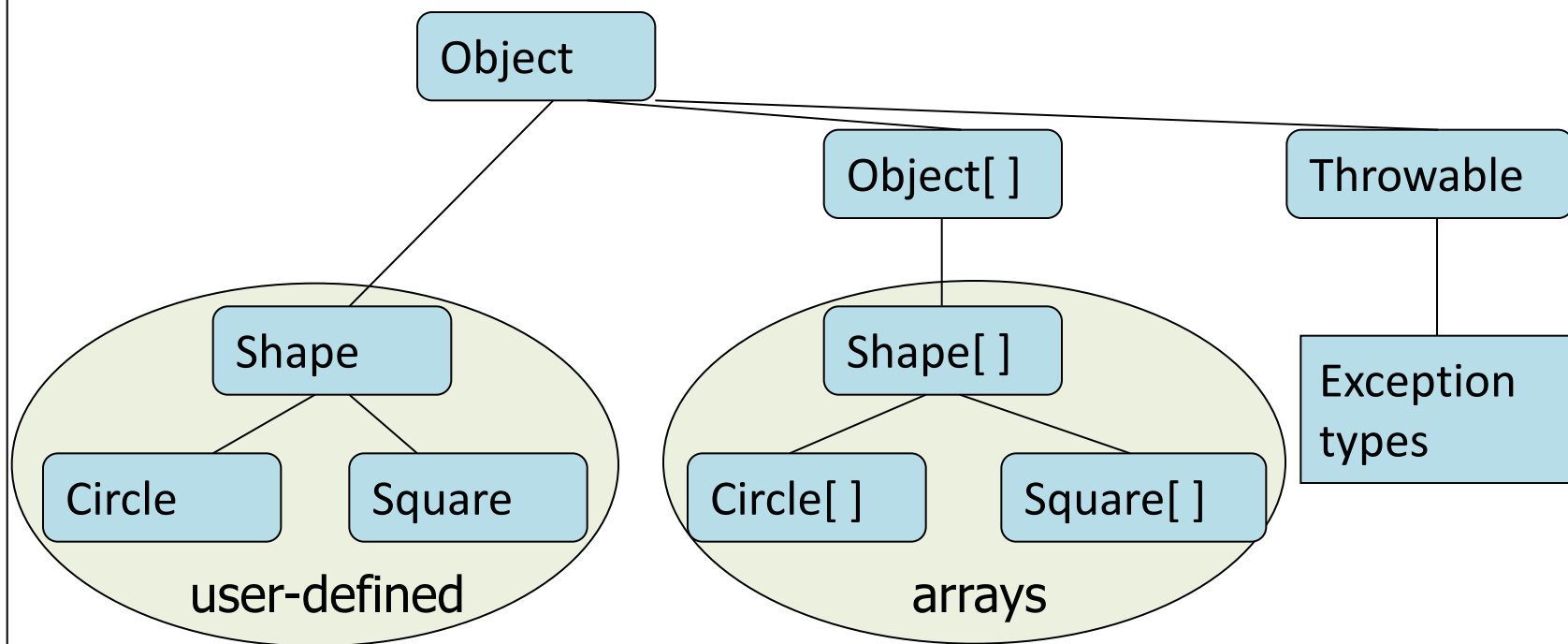
- In principle: `ColorPoint <: Point`
- In practice: This is covariant case; contravariance is another story

Java Types

- Two general kinds of types
 - Primitive types – *not* objects
 - Integers, Booleans, etc
 - Reference types
 - Classes, interfaces, arrays
 - No syntax distinguishing `Object *` from `Object`
- Static type checking
 - Every expression has type, determined from its parts
 - Some auto conversions, many casts are checked at run time
 - Example, assuming `A <: B`
 - If `A x`, then can use `x` as argument to method that requires `B`
 - If `B x`, then can try to cast `x` to `A`
 - Downcast checked at run-time, may raise exception

Classification of Java Types

Reference Types



Primitive Types



Subtyping

- Primitive types
 - Conversions: int -> long, double -> long, ...
- Class subtyping similar to C++
 - Subclass produces subtype
 - Single inheritance => subclasses form tree
- Interfaces
 - Completely abstract classes
 - no implementation
 - Multiple subtyping
 - Interface can have multiple subtypes (implements, extends)
- Arrays
 - Covariant subtyping – not consistent with semantic principles

Java Class Subtyping

- Signature Conformance
 - Subclass method signatures must conform to superclass
- Three ways signature could vary
 - Argument types
 - Return type
 - Exceptions
- Java rule
 - Java 1.1: Arguments and returns must have identical types, may remove exceptions
 - Java 1.5: covariant return type specialization

Interface Subtyping: Example

```
interface Shape {  
    public float center();  
    public void rotate(float degrees);  
}  
  
interface Drawable {  
    public void setColor(Color c);  
    public void draw();  
}  
  
class Circle implements Shape, Drawable {  
    // does not inherit any implementation  
    // but must define Shape, Drawable methods  
}
```

Q: can interfaces be recursive?

Properties of Interfaces

- Flexibility
 - Allows subtype graph instead of tree
 - Avoids problems with multiple inheritance of implementations (remember C++ “diamond”)
- Cost
 - Offset in method lookup table not known at compile
 - Different bytecodes for method lookup
 - one when class is known
 - one when only interface is known
 - search for location of method
 - cache for use next time this call is made (from this line)

Array Types

- Automatically defined
 - Array type `T[]` exists for each class, interface type `T`
 - Cannot extend array types (array types are final)
 - Multi-dimensional arrays are arrays of arrays: `T[][]`
- Treated as reference type
 - An array variable is a pointer to an array, can be null
 - Example: `Circle[] x = new Circle[array_size]`
 - Anonymous array expression: `new int[] {1,2,3, ... 10}`
- Every array type is a subtype of `Object[]`, `Object`
 - Length of array is not part of its static type

Array Subtyping

- Covariance
 - if $S <: T$ then $S[] <: T[]$

- Standard type error

```
class A {...}
```

```
class B extends A {...}
```

```
B[] bArray = new B[10]
```

```
A[] aArray = bArray // considered OK since B[] <: A[]
```

```
aArray[0] = new A() // compiles, but run-time error
```

```
// raises ArrayStoreException
```

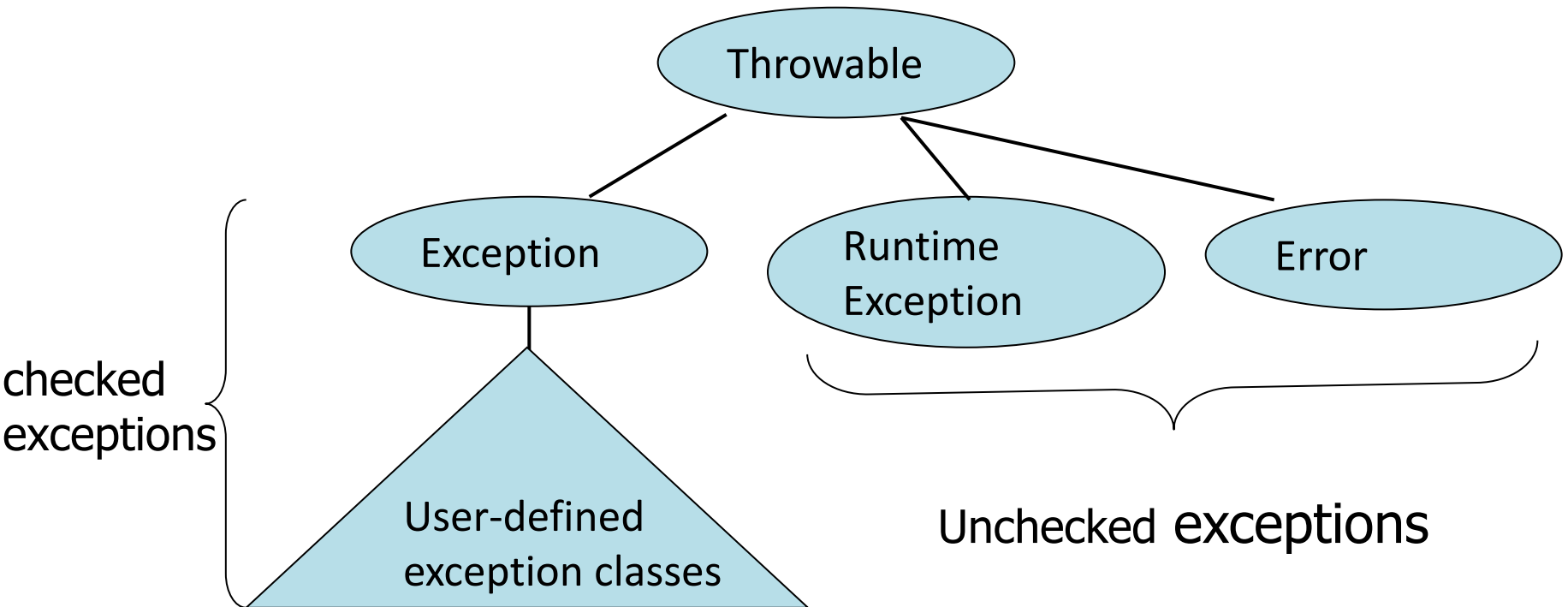
Covariance problem again ...

- Simula problem
 - If $A <: B$, then $A \text{ ref} <: B \text{ ref}$
 - Needed run-time test to prevent bad assignment
 - Covariance for assignable cells is not right in principle
- Explanation
 - interface of “T reference cell” is
 - $\text{put} : T \rightarrow T \text{ ref}$
 - $\text{get} : T \text{ ref} \rightarrow T$
 - Remember covariance/contravariance of functions

Java Exceptions

- Similar basic functionality to other languages
 - Constructs to *throw* and *catch* exceptions
 - Dynamic scoping of handler
- Some differences
 - An exception is an object from an exception class
 - Subtyping between exception classes
 - Use subtyping to match type of exception or pass it on ...
 - Similar functionality to ML pattern matching in handler
 - Type of method includes exceptions it can throw
 - Actually, only subclasses of Exception (see next slide)

Exception Classes



If a method may throw a checked exception, then exception must be in the type of the method

Why define new exception types?

- Exception may contain data
 - Class Throwable includes a string field so that cause of exception can be described
 - Pass other data by declaring additional fields or methods
- Subtype hierarchy used to catch exceptions

`catch <exception-type> <identifier> { ... }`

will catch any exception from any subtype of exception-type and bind object to identifier

Subtyping concepts

- Type of an object represents its interface
- Subtyping has associated substitution principle
 - If $A \leq B$, then A objects can be used in place of B objects
- Implicit subtyping in dynamically typed lang
 - Relation between interfaces determines substitutivity
- Explicit subtyping in statically typed languages
 - Type checker may recognize some subtyping
 - Issues: programming style, implementation efficiency
- Covariance and contravariance
 - Function argument types *reverse* order
 - Problems with Java array covariance

Principles

- Object “width” subtyping
- Function covariance, contravariance
- Object type “depth” subtyping
- Subtyping recursive types

Applications of Principles

- Dynamically typed languages
 - If $A <: B$ in principle, then can use A objects in place of B objects
- C++
 - Class subtyping only when public base class
 - Compiler allows width subtyping, covariant depth subtyping. (Think about why...)
- Java
 - Class subtyping only when declared using “extends”
 - Class and interface subtyping when declared
 - Compiler allows width subtyping, covariant depth subtyping
 - Additional typing issues related to generics

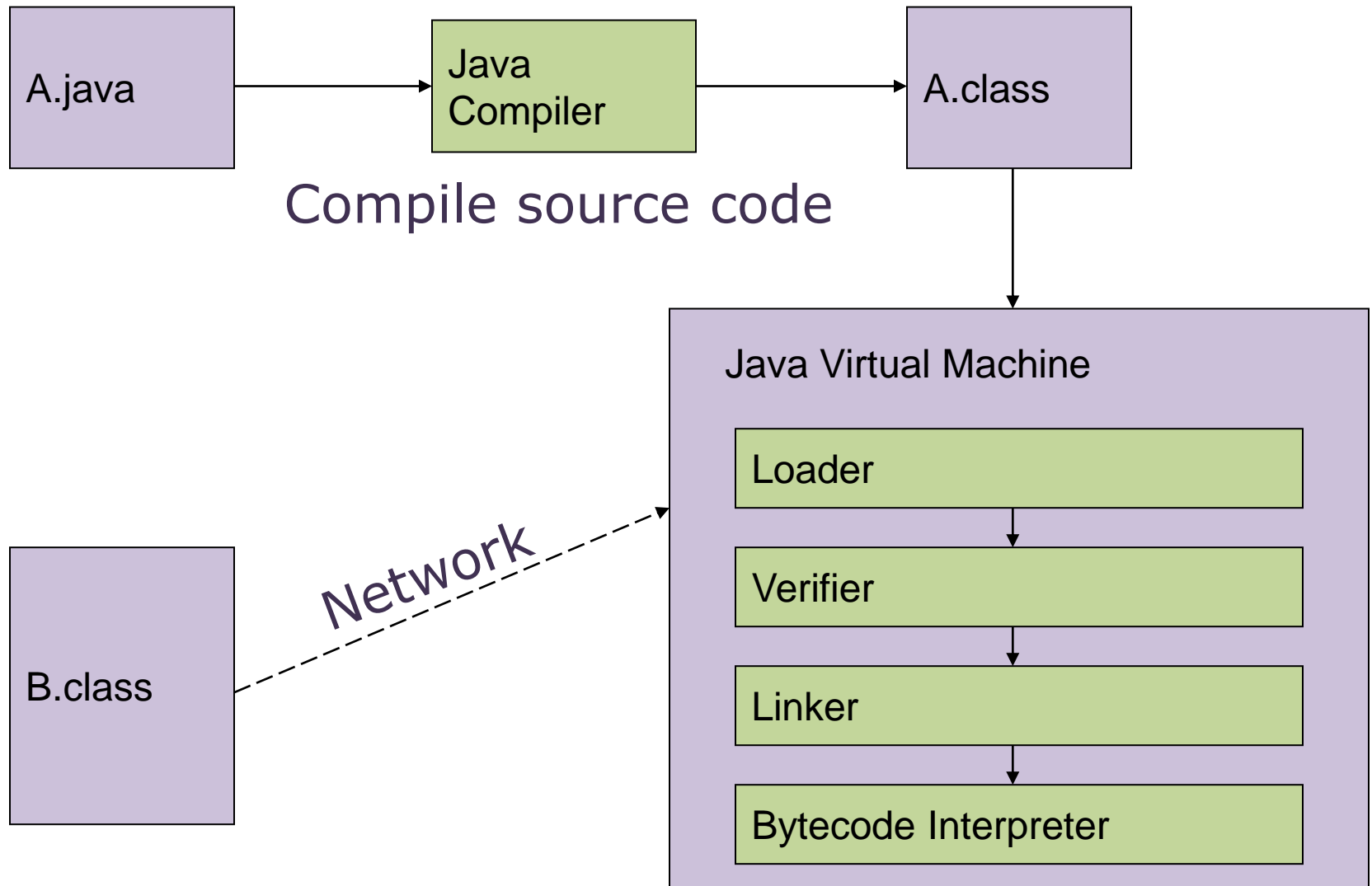
Java Language Implementation: Outline

- Java virtual machine overview
 - Loader and initialization
 - Linker and verifier
 - Bytecode interpreter
- JVM Method lookup
 - four different bytecodes
- Verifier analysis
- Method lookup optimizations (beyond Java)
- Java security
 - Buffer overflow
 - Java “sandbox”
 - Stack inspection

Java Implementation

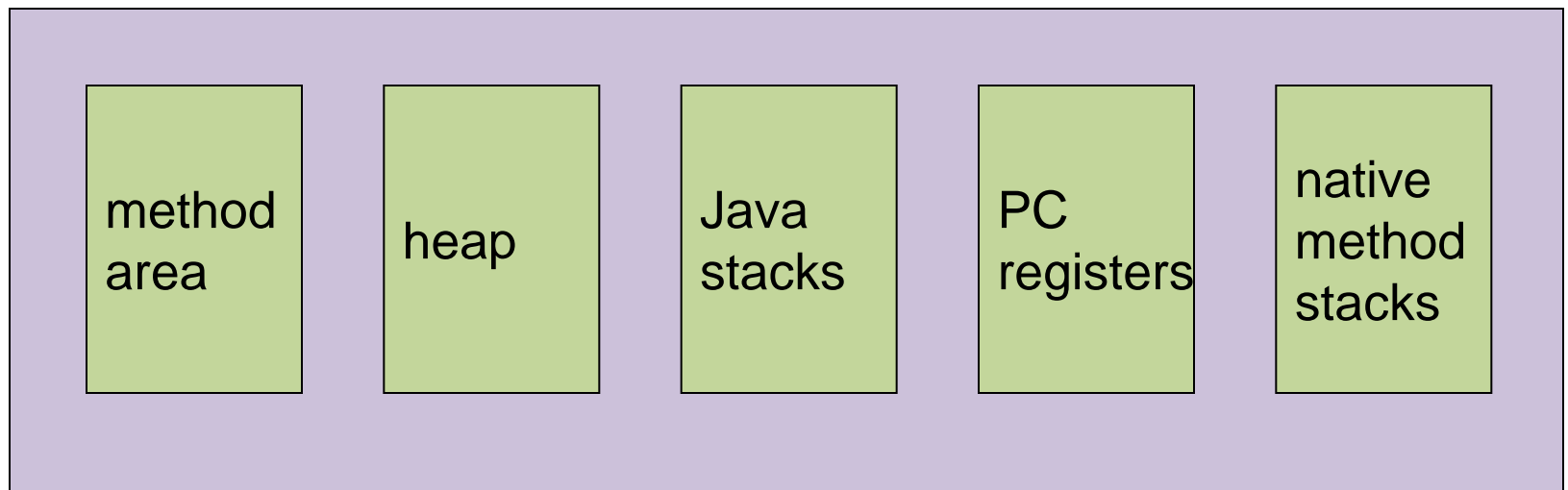
- Compiler and Virtual Machine
 - Compiler produces bytecode
 - Virtual machine loads classes on demand, verifies bytecode properties, interprets bytecode
- Why this design?
 - Bytecode interpreter/compilers used before
 - Pascal “pcode”; Smalltalk compilers use bytecode
 - Minimize machine-dependent part of implementation
 - Do optimization on bytecode when possible
 - Keep bytecode interpreter simple
 - For Java, this gives portability
 - Transmit bytecode across network

Java Virtual Machine Architecture



JVM Memory Areas

- Java program has one or more threads
- Each thread has its own stack
- All threads share same heap



Class Loader

- Runtime system loads classes as needed
 - When class is referenced, loader searches for file of compiled bytecode instructions
- Default loading mechanism can be replaced
 - Define alternate ClassLoader object
 - Extend the abstract ClassLoader class and implementation
 - ClassLoader does not implement abstract method loadClass, but has methods that can be used to implement loadClass
 - Can obtain bytecodes from alternate source
 - VM restricts applet communication to site that supplied applet

Example issue in class loading and linking:

Static members and initialization

```
class ... {  
    /* static variable with initial value */  
    static int x = initial_value  
    /* ---- static initialization block      --- */  
    static { /* code executed once, when loaded */ }  
}
```

- Initialization is important
 - Cannot initialize class fields until loaded
- Static block cannot raise an exception
 - Handler may not be installed at class loading time

JVM Linker and Verifier

- Linker
 - Adds compiled class or interface to runtime system
 - Creates static fields and initializes them
 - Resolves names
 - Checks symbolic names and replaces with direct references
- Verifier
 - Check bytecode of a class or interface before loaded
 - Throw `VerifyError` exception if error occurs

Verifier

- Bytecode may not come from standard compiler
 - Evil hacker may write dangerous bytecode
- Verifier checks correctness of bytecode
 - Every instruction must have a valid operation code
 - Every branch instruction must branch to the start of some other instruction, not middle of instruction
 - Every method must have a structurally correct signature
 - Every instruction obeys the Java type discipline

Last condition is complicated.

Bytecode Interpreter

- Standard virtual machine interprets instructions
 - Perform run-time checks such as array bounds
 - Possible to compile bytecode class file to native code
- Java programs can call native methods
 - Typically functions written in C
- Multiple bytecodes for method lookup
 - `invokevirtual` - when class of object known
 - `invokeinterface` - when interface of object known
 - `invokestatic` - static methods
 - `invokespecial` - some special cases

Type Safety of JVM

- Run-time type checking
 - All casts are checked to make sure type safe
 - All array references are checked to make sure the array index is within the array bounds
 - References are tested to make sure they are not null before they are dereferenced
- Additional features
 - Automatic garbage collection
 - No pointer arithmetic

If program accesses memory, that memory is allocated to the program and declared with correct type

JVM Uses Stack Machine

- Java

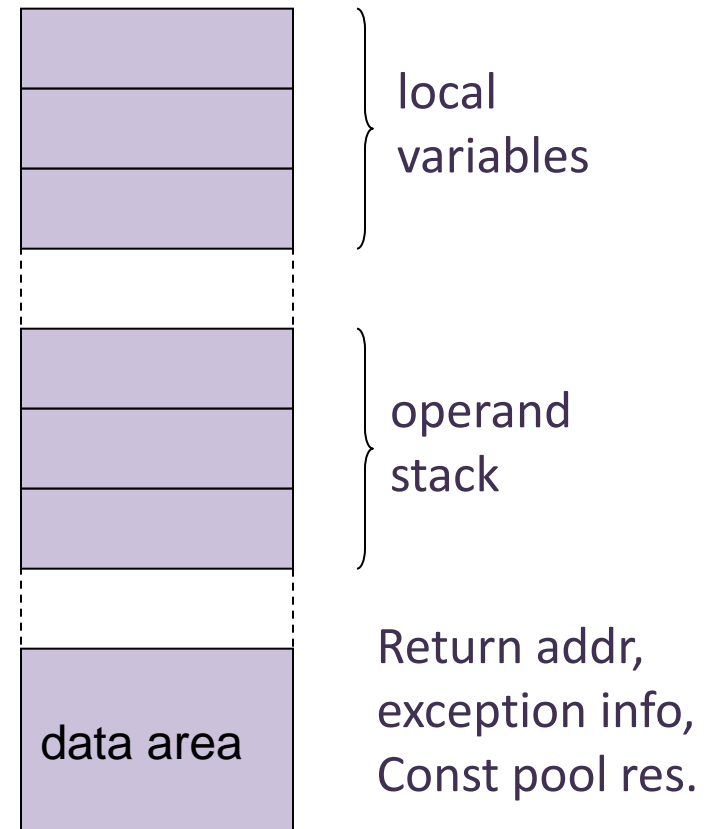
```
Class A extends Object {  
    int i  
    void f(int val) { i = val + 1;}  
}
```

- Bytecode

```
Method void f(int)  
    aload 0 ; object ref this  
    iload 1 ; int val  
    iconst 1  
    iadd ; add val +1  
    putfield #4 <Field int i>  
    return
```

↑
refers to constant pool

JVM Activation Record



Field and Method Access

- Instruction includes index into constant pool
 - Constant pool stores symbolic names
 - Store once, instead of each instruction, to save space
- First execution
 - Use symbolic name to find field or method
- Second execution
 - Use modified “quick” instruction to simplify search

Outline

- Java virtual machine overview
 - Loader and initialization
 - Linker and verifier
 - Bytecode interpreter
- JVM Method lookup
 - four different bytecodes
- Verifier analysis
- Method lookup optimizations (beyond Java)
- Java security
 - Buffer overflow
 - Java “sandbox”
 - Stack inspection

Java Security

- Security
 - Prevent unauthorized use of computational resources
- Java security
 - Java code can read input from careless user or malicious attacker
 - Java code can be transmitted over network – code may be *written* by careless friend or malicious attacker

Java is designed to reduce many security risks

Java Security Mechanisms

- Sandboxing
 - Run program in restricted environment
 - Analogy: child's sandbox with only safe toys
 - This term refers to
 - Features of loader, verifier, interpreter that restrict program
 - Java Security Manager, a special object that acts as access control "gatekeeper"
- Code signing
 - Use cryptography to establish origin of class file
 - This info can be used by security manager

Buffer Overflow Attack

- Most prevalent *general* security problem today
 - Large number of CERT advisories are related to buffer overflow vulnerabilities in OS, other code
- General network-based attack
 - Attacker sends carefully designed network msgs
 - Input causes privileged program (e.g., Sendmail) to do something it was not designed to do
- Does not work in Java
 - Illustrates what Java was designed to prevent

Sample C code to illustrate attack

```
void f (char *str) {
    char buffer[16];
    ...
    strcpy(buffer,str);
}

void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
        large_string[i] = 'A';
    f(large_string);
}
```

- Function
 - Copies str into buffer until null character found
 - Could write past end of buffer, *over function return addr*
- Calling program
 - Writes 'A' over f activation record
 - Function f “returns” to location 0x4141414141
 - This causes segmentation fault
- Variations
 - Put meaningful address in string
 - Put *code* in string and jump to it !!

Java Sandbox

- Four complementary mechanisms
 - Class loader
 - Separate namespaces for separate class loaders
 - Associates *protection domain* with each class
 - Verifier and JVM run-time tests
 - NO unchecked casts or other type errors, NO array overflow
 - Preserves private, protected visibility levels
 - Security Manager
 - Called by library functions to decide if request is allowed
 - Uses protection domain associated with code, user policy
 - Coming up in a few slides: stack inspection

Security Manager

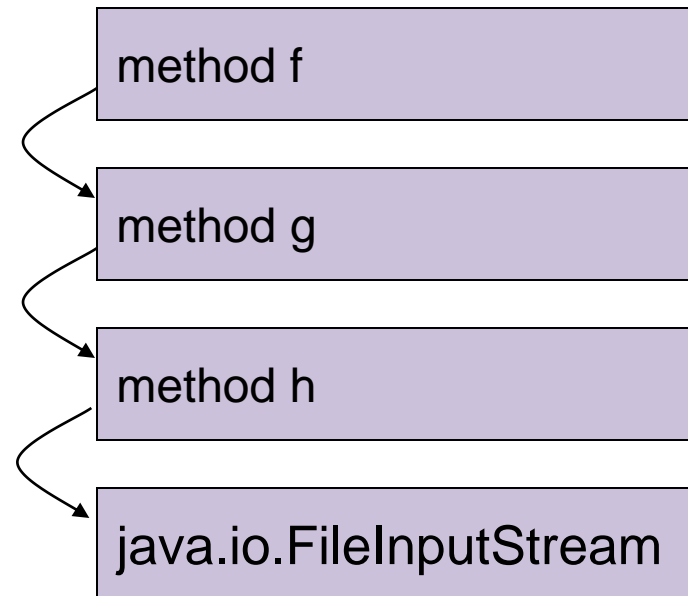
- Java library functions call security manager
- Security manager object answers at run time
 - Decide if calling code is allowed to do operation
 - Examine protection domain of calling class
 - Signer: organization that signed code before loading
 - Location: URL where the Java classes came from
 - Uses the system policy to decide access permission

Sample SecurityManager methods

checkExec	Checks if the system commands can be executed.
checkRead	Checks if a file can be read from.
checkWrite	Checks if a file can be written to.
checkListen	Checks if a certain network port can be listened to for connections.
checkConnect	Checks if a network connection can be created.
checkCreateClassLoader	Check to prevent the installation of additional ClassLoaders.

Stack Inspection

- Permission depends on
 - Permission of calling method
 - Permission of all methods above it on stack
 - Up to method that is trusted and asserts this trust



Example: privileged printing

```
privPrint(f) = (* owned by system *)  
{  
    checkPrivilege(PrintPriv);  
    print(f);  
}
```

```
foreignProg() = (* owned by Joe *)  
{  
    ...; privPrint(file); ...;  
}
```

Stack Inspection

- Stack frames are annotated with names of owners and any enabled privileges
- During inspection, stack frames are searched from most to least recent:
 - **fail** if a frame belonging to someone not authorized for privilege is encountered
 - **succeed** if activated privilege is found in frame