CS 242 2012

Object typing and subtypes

Reading

Chapter 10, section 10.2.3
Chapter 11, sections 11.3.2 and 11.7
Chapter 12, section 12.4
Chapter 13, section 13.3

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

<table>
<tr><td>class name</td><td>ColorPoint</td></tr>
<tr><td>super class</td><td>Point</td></tr>
<tr><td>class var</td><td></td></tr>
<tr><td>instance var</td><td>color</td></tr>
<tr><td colspan="2">class messages and methods</td></tr>
<tr><td>newX:xv Y:yv C:cv</td><td>〈 … code … 〉</td></tr>
<tr><td colspan="2">instance messages and methods</td></tr>
<tr><td>color</td><td>| | ^color</td></tr>
<tr><td>draw</td><td>〈 … code … 〉</td></tr>
</table>

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	Colored_point
x:y:	x:y:
moveDx:Dy:	moveDx:Dy:
x	x
y	y
draw	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

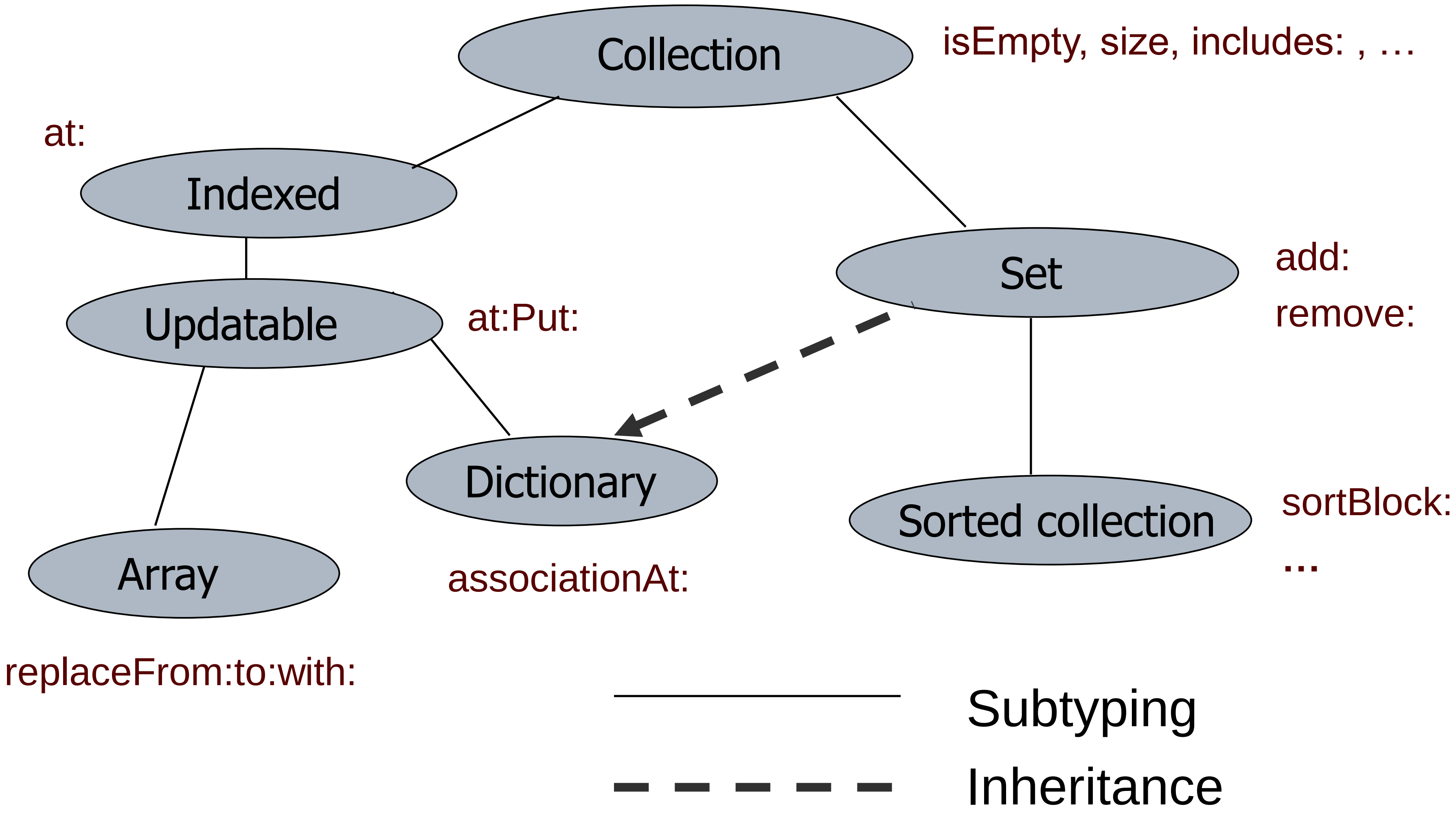
- Subtyping
  - If interface is superset, then a subtype
  - Example: ColorPoint subtype of Point
  - Sometimes called “conformance”

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();
void move(int);

Public members

ColorPoint: int getX();
int getColor();
void move(int);
void darken(int tint);

Public members

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

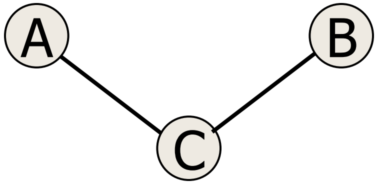
Implementation of subtyping

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

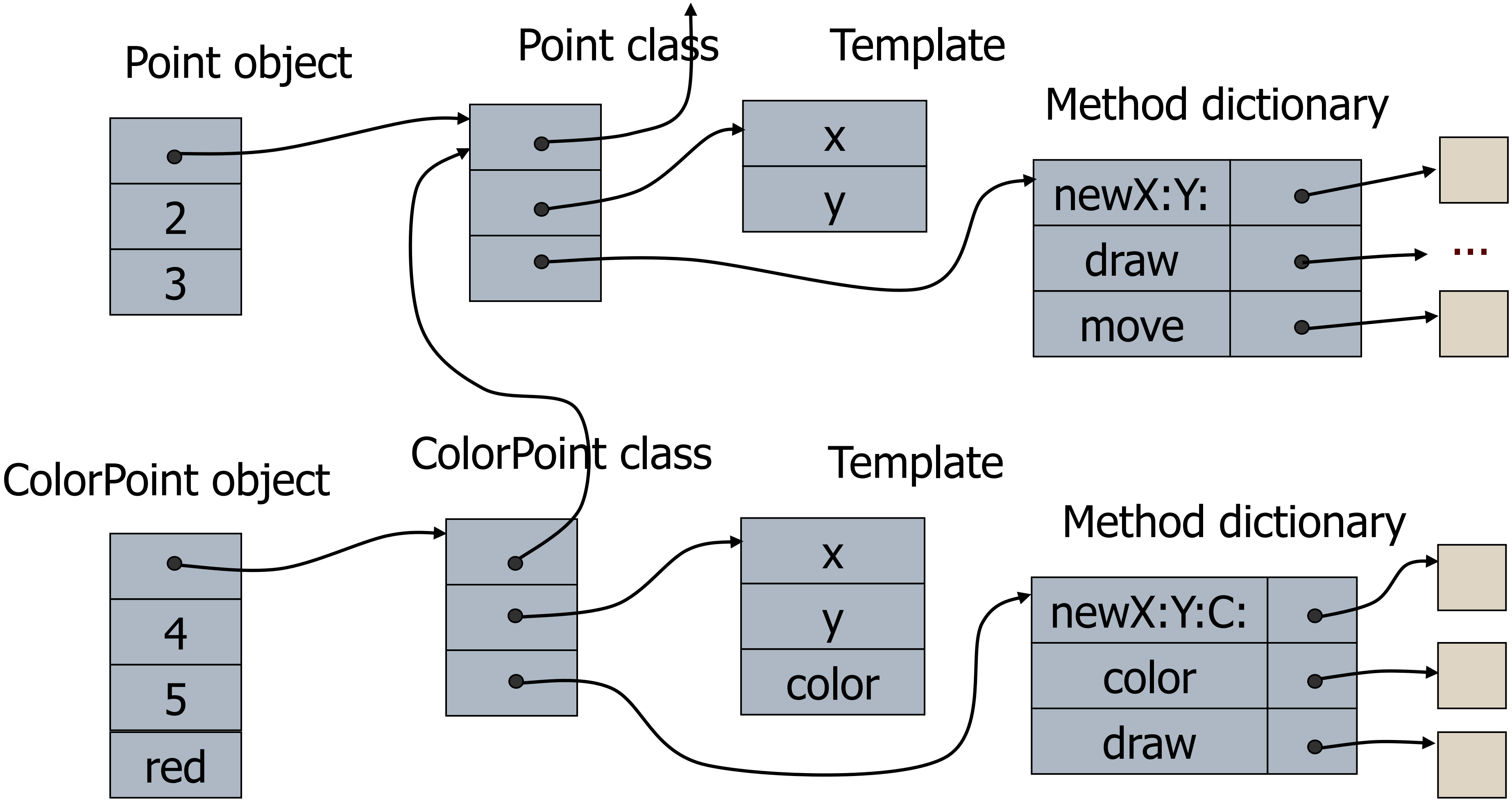
```
circle *c = new Circle(p,r);
shape *s = c;                 // s points to circle c
```

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

```
C *pc = new C;
B *pb = pc;
A *pa = pc;
// may point to different position in object
```

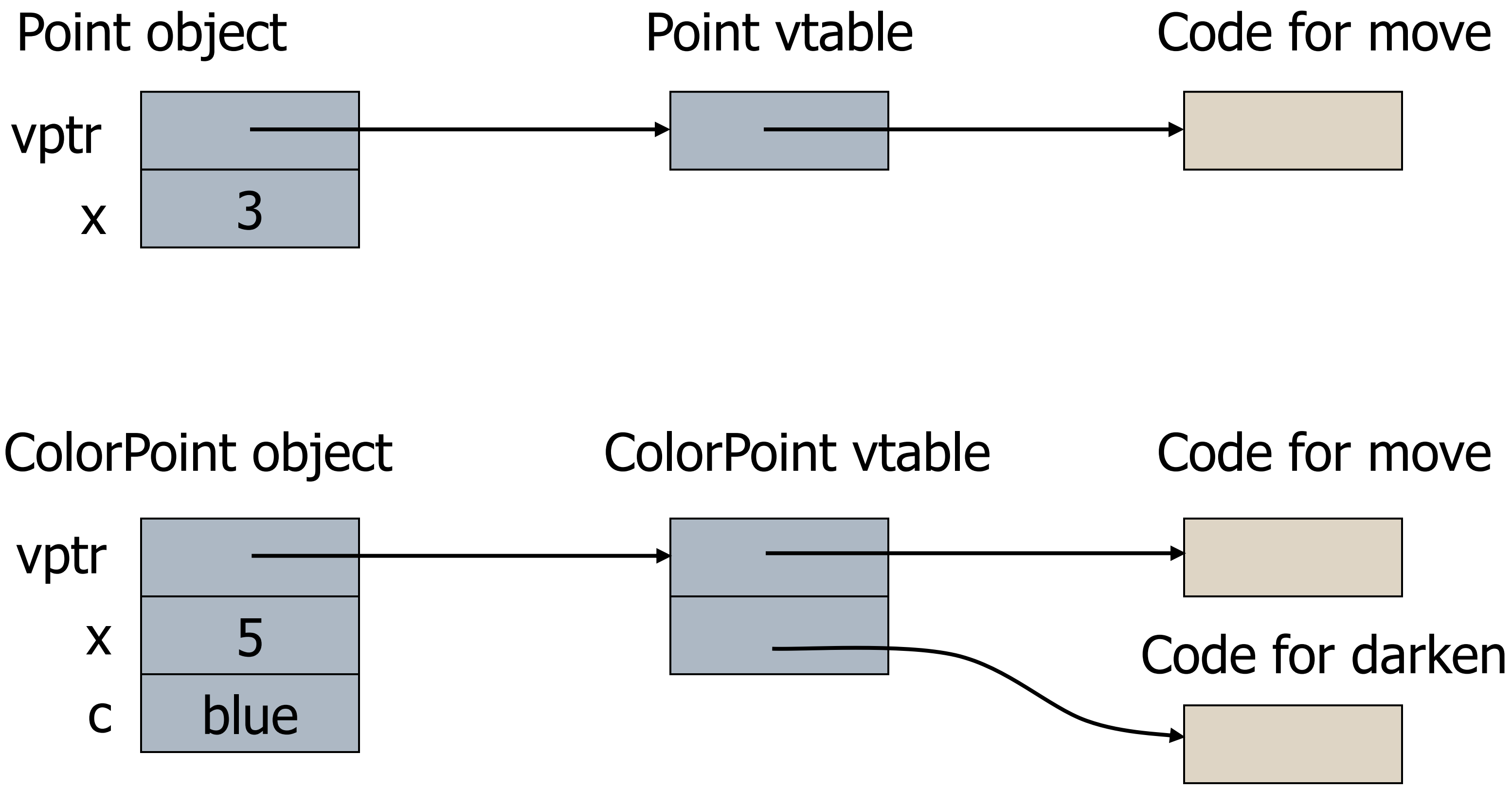


Smalltalk/JavaScript Representation



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

C++ Run-time representation



Point object

vptr

x 3

Point vtable

Code for move

ColorPoint object

vptr

x 5

c blue

ColorPoint vtable

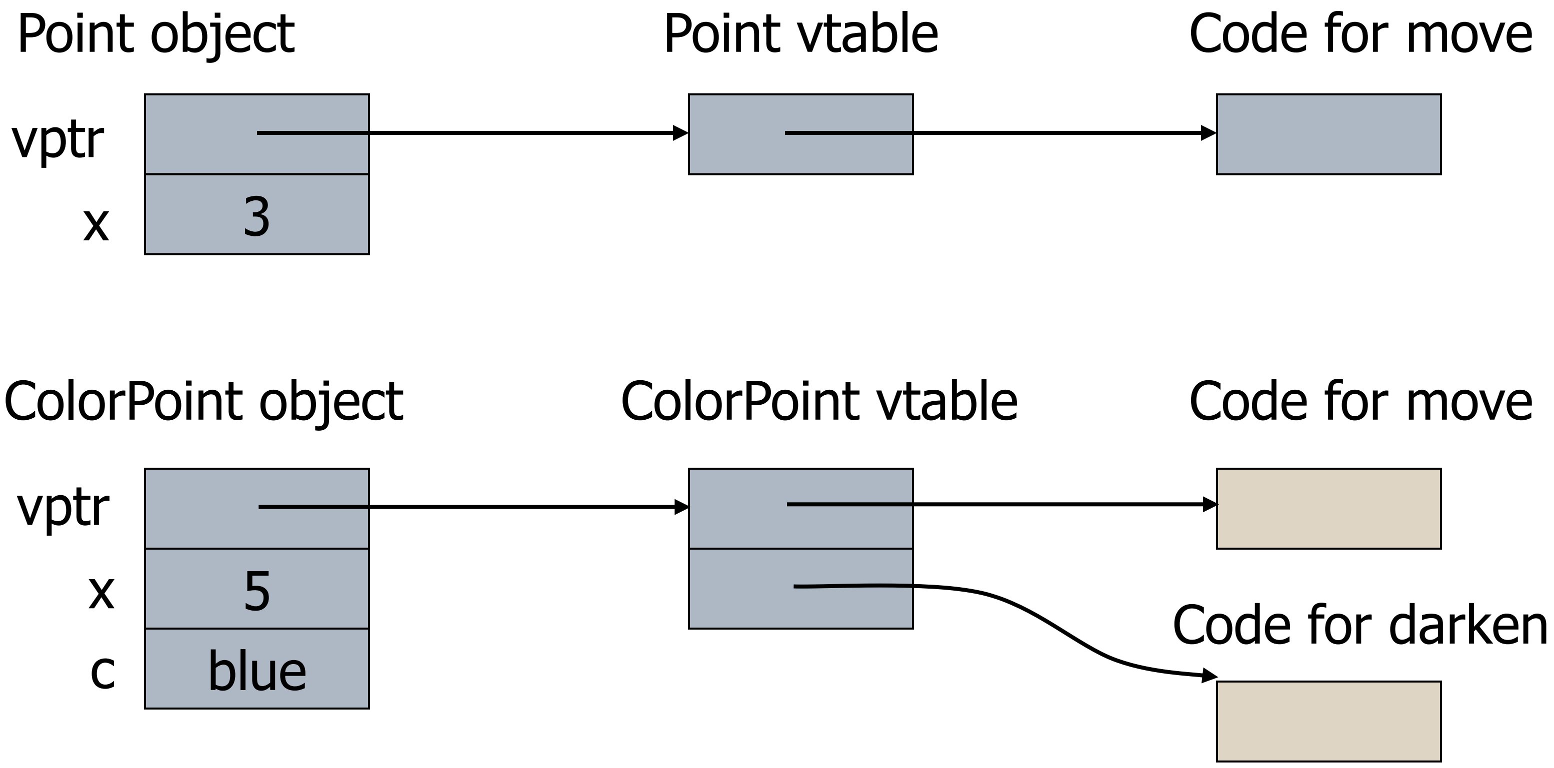
Code for move

Code for darken

Data at same offset

Function pointers at same offset

C++: virtual function lookup



Point object

vptr

x

3

Point vtable

Code for move

ColorPoint object

vptr

x

5

c

blue

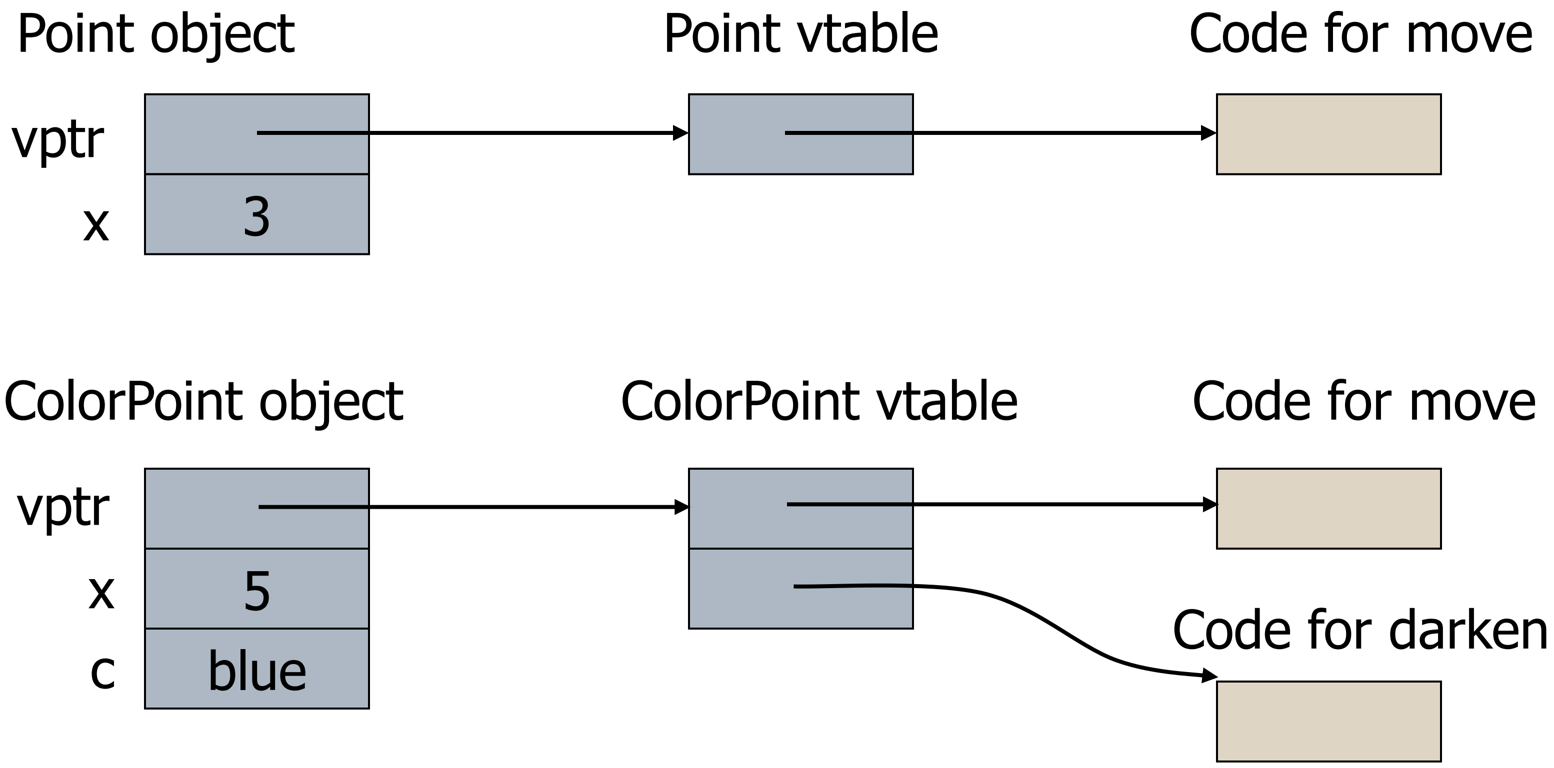
ColorPoint vtable

Code for move

Code for darken

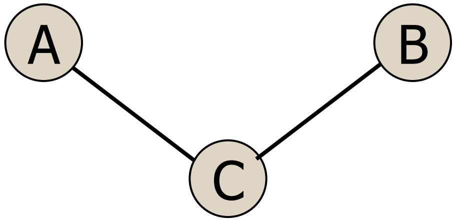
```
Point p = new Pt(3);
p->move(2);          // (*(p->vptr[0]))(p,2)
```

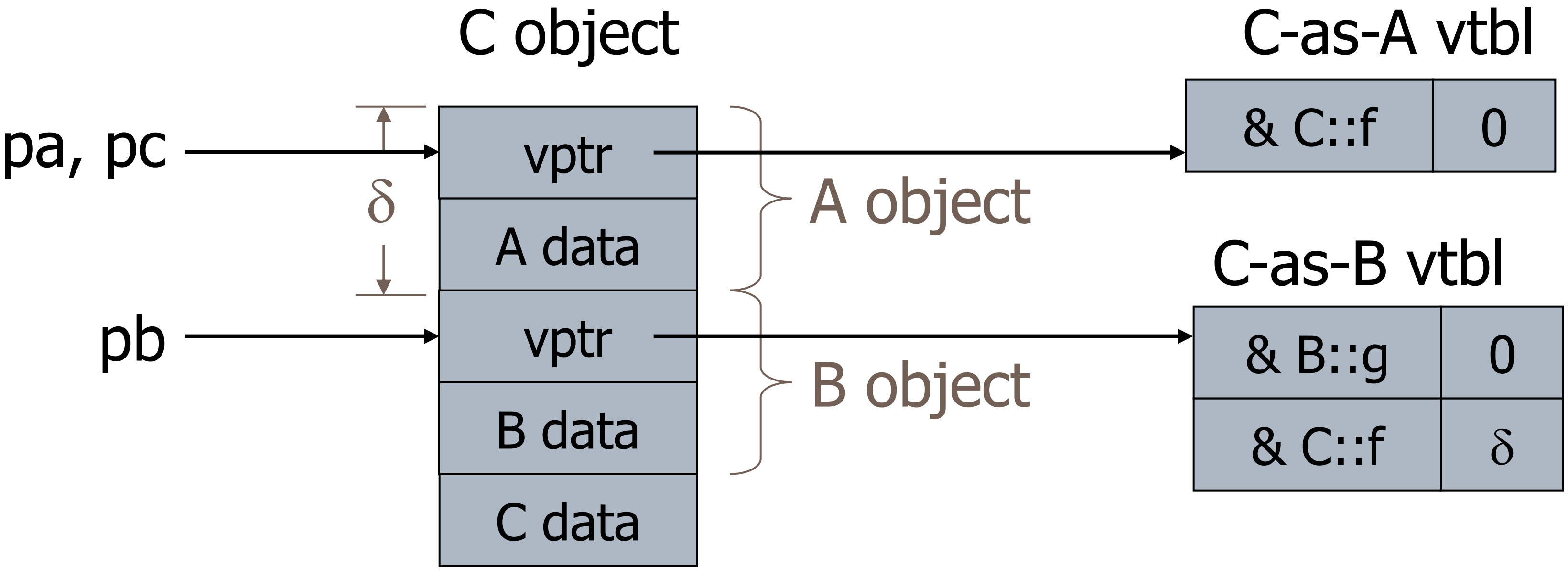
C++: virtual function lookup, part 2



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

C++ Multiple Inheritance





C object

pa, pc

pb

δ

vptr
A data
vptr
B data
C data

A object

B object

C-as-A vtbl

& C::f	0

C-as-B vtbl

& B::g	0
& C::f	δ

- 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:     ...
};
```

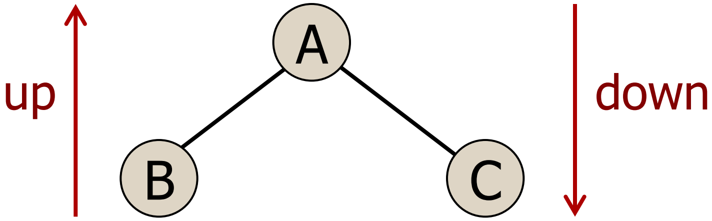
- C++ does not treat ColorPoint <: Point   as written
  - Need public inheritance ColorPoint : public Point
  - Why??

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:



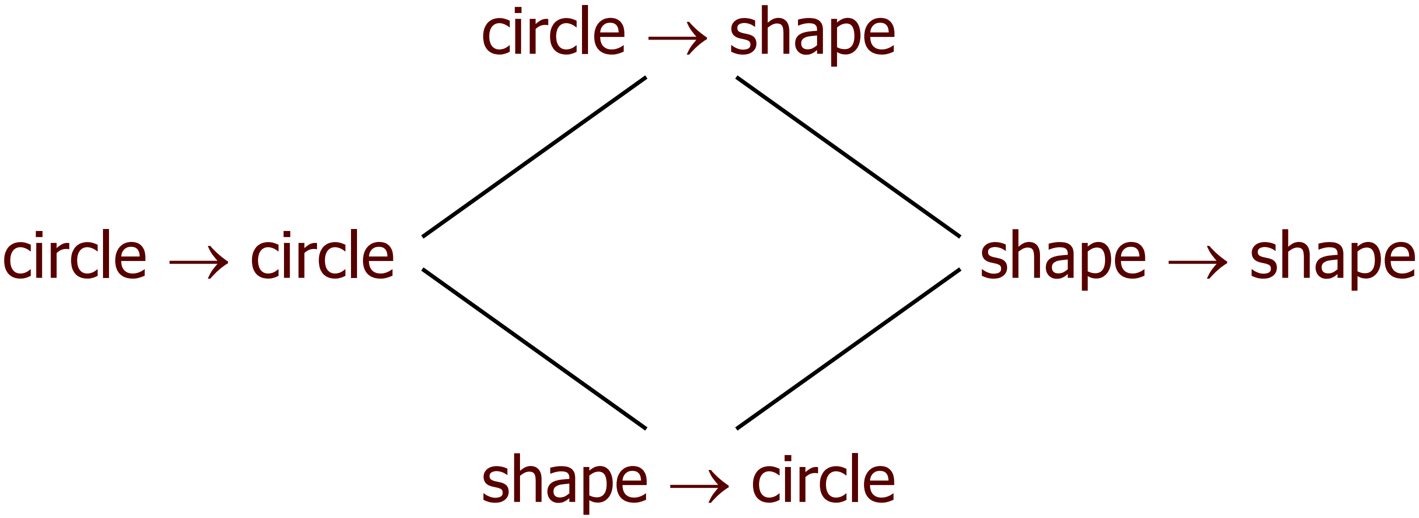
```
class A(…);  …
A class B(…); …
ref (A) a :- new A(…)
ref (B) b :- new B(…)
a := b        /* OK since B is subclass of A */
…
b := a        /* compiles, but run-time test  */
```

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 $C \to A \;<:\; C \to B$
- Subtyping for function arguments
  - If $A <: B$, then $B \to C \;<:\; A \to C$
- Terminology
  - Covariance: $A <: B$ implies $F(A) \;<:\; F(B)$
  - Contravariance: $A <: B$ implies $F(B) \;<:\; F(A)$

Examples

- If circle <: shape, then



circle → shape

circle → circle

shape → shape

shape → circle

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

Subtyping principles (recap)

- "Width" subtyping for object types

$$\frac{i > j}{[m_1{:}\pi_1, \ldots, m_i{:}\ \pi_i] <: [m_1{:}\pi_1, \ldots, m_j{:}\ \pi_j]}$$

i

j

- "Depth" subtyping for object types

$$\frac{\sigma_i <: \pi_i}{[m_1{:}\ \sigma_1, \ldots, m_i{:}\ \sigma_J] <: [m_1{:}\pi_1, \ldots, m_j{:}\ \pi_j]}$$

- Function subtyping

$$\frac{\sigma' <: \sigma \qquad \pi <: \pi'}{\sigma \to \pi \ <: \ \sigma' \to \pi'}$$

Subtyping recursive types

- Principle

$$\frac{s <: t \;\; \Rightarrow \;\; \sigma(s) <: \pi(t)}{\text{type } s = \sigma(s) \;\; <: \text{type } t = \pi(t)}$$

s not in $\pi(t)$
t not in $\sigma(s)$

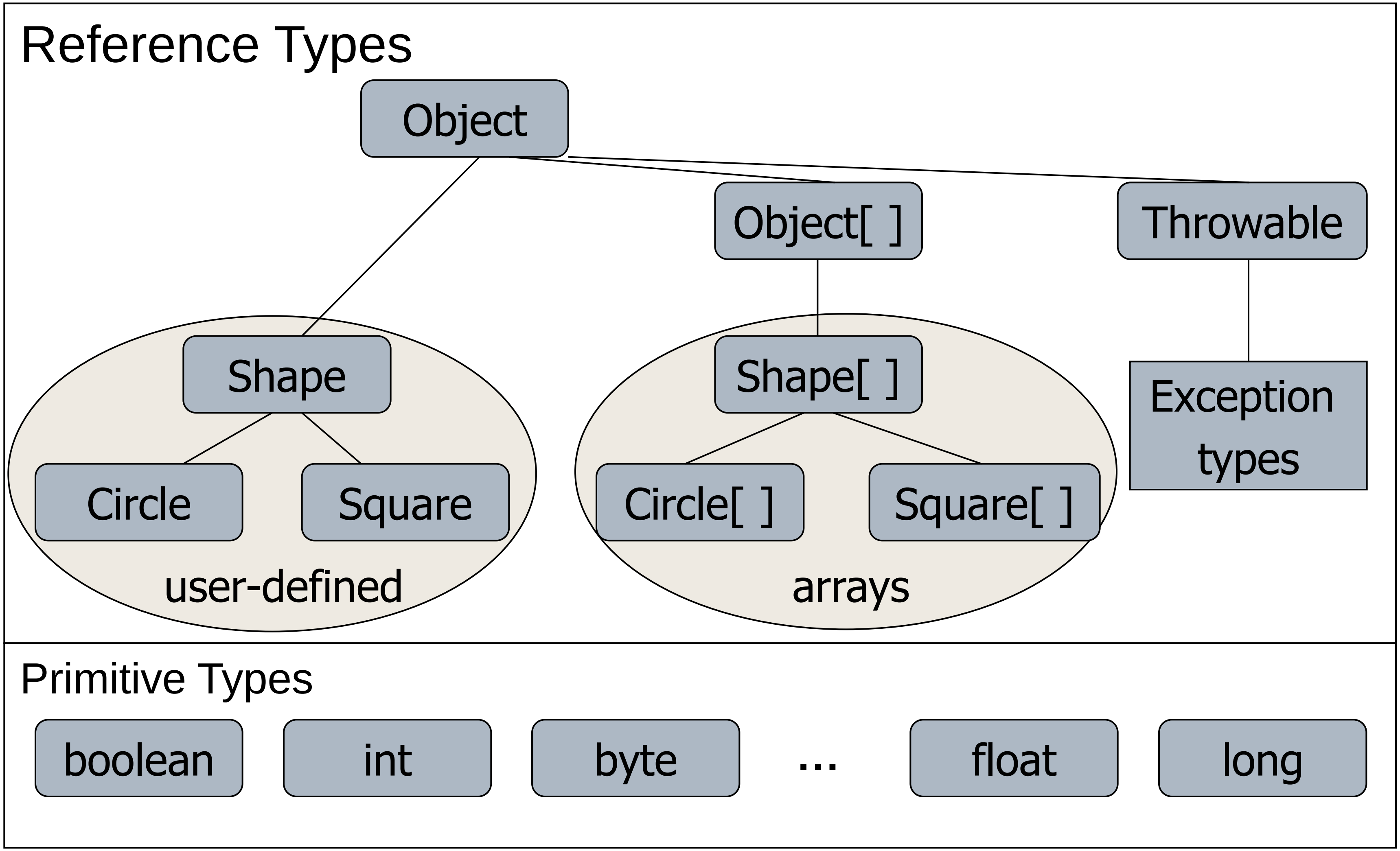
- Example

$$\frac{\text{cp} <: \text{p} \Rightarrow \{ \ldots \text{ mv: int} \to \text{cp}\} <: \{ \ldots \text{ mv: int} \to \text{p} \}}{\text{type cp} = \{ \ldots \text{ mv: int} \to \text{cp}\} <: \text{type p} = \{ \ldots \text{ mv: int} \to \text{p}\}}$$

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

Object

Object[ ]

Throwable

Shape

Circle

Square

user-defined

Shape[ ]

Circle[ ]

Square[ ]

arrays

Exception types

Primitive Types

boolean

int

byte

…

float

long

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
  - How much conformance is needed in principle?
- 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)

More about this later …

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 ref <: B 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

    put : $T \rightarrow T\ ref$

    get : $T\ ref \rightarrow T$
  - Remember covariance/contravariance of functions

Afterthought on Java arrays

Date: Fri, 09 Oct 1998 09:41:05 -0600

From: bill joy

Subject: …[discussion about java genericity]

actually, java array covariance was done for less noble reasons …: it made some generic "bcopy" (memory copy) and like operations much easier to write...

I proposed to take this out in 95, but it was too late (...).

i think it is unfortunate that it wasn't taken out...

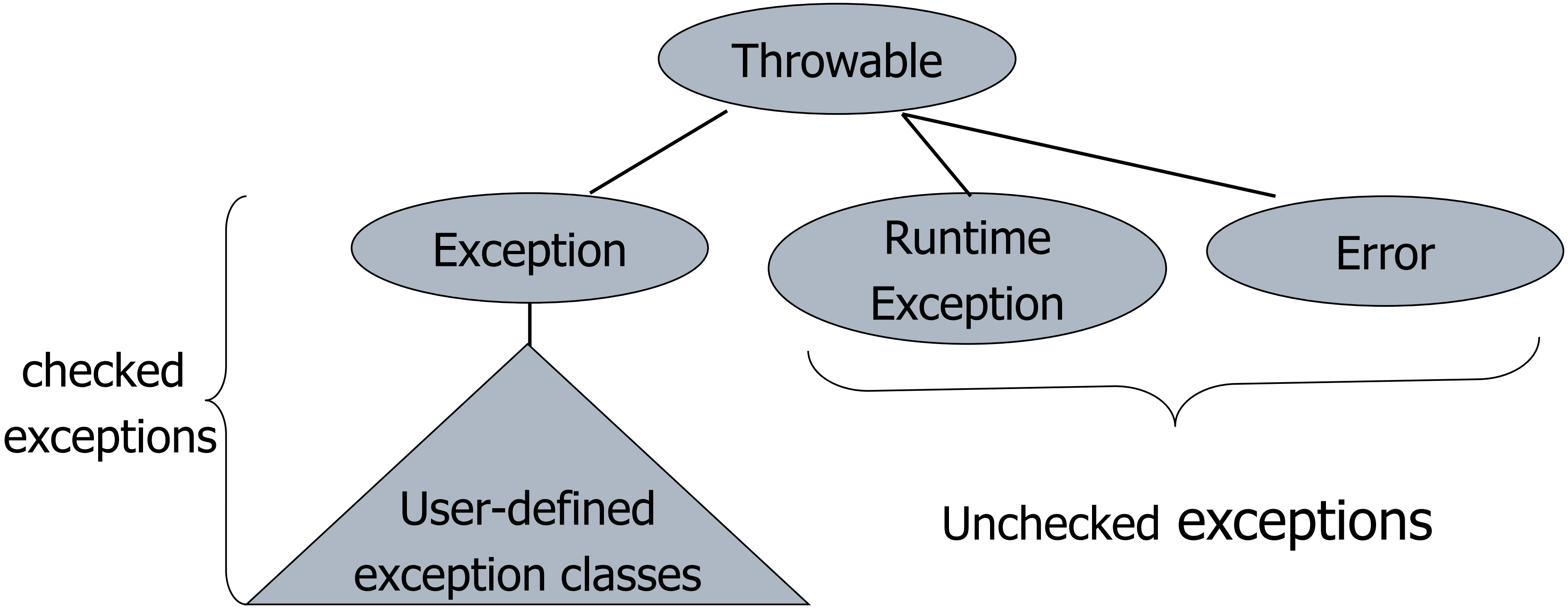
it would have made adding genericity later much cleaner, and [array covariance] doesn't pay for its complexity today.

wnj

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 <: 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