Languages and Compilers (SProg og Oversættere)

Lecture 19

Abstract Data Types and Object Oriented Features

Bent Thomsen

Department of Computer Science

Aalborg University

With acknowledgement to John Mitchell, Elsa Gunter, David Watt and Amiram Yehudai whose slides this lecture is based on.

Learning goals

- To understand the concept of abstract data types
- Understand implementations of abstract data types
- Understand concepts of Object Oriented programming:
  - Classes and objects
  - Inheritance
  - Dynamic dispatch
- Understand how classes and objects can be implemented
- Understand issues in modularity of large programs

Tennent's Language Design principles

- The Principle of Abstraction
  - All major syntactic categories should have abstractions defined over them. For example, functions are abstractions over expressions
- The Principle of Correspondence
  - Declarations ≈ Parameters
- The Principle of Data Type Completeness
  - All data types should be first class without arbitrary restriction on their use

–Originally defined by R.D.Tennent

The Concept of Abstraction

- The concept of abstraction is fundamental in programming (and computer science)
- Tennents principle of abstraction
  - is based on identifying all of the semantically-meaningful syntactic categories of the language and then designing a coherent set of abstraction facilities for each of these.
- Nearly all programming languages support process (or command) abstraction with subprograms (procedures)
- Many programming languages support expression abstraction with functions
- Nearly all programming languages designed since 1980 have supported data abstraction:
  - Abstract data types
  - Objects
  - Modules

What have we seen so far?

- Structured data
  - Arrays
  - Records or structs
  - Lists
- Visibility of variables and subprograms
  - Scope rules
- Why is this not enough?

Information Hiding

- Consider the C code:

```
typedef struct RationalType {
   int numerator;
   int denominator;
} Rational

Rational mk_rat (int n,int d) { …}
Rational add_rat (Rational x, Rational y) {
… }
```

- Can use **`mk_rat`**, **`add_rat`** without knowing the details of **`RationalType`**

Need for Abstract Types

- Problem: abstraction not enforced
  - User can create Rationals without using **`mk_rat`**
  - User can access and alter numerator and denominator directly without using provided functions

- With abstraction we also need information hiding

Abstract Types - Example

- Suppose we need sets of integers
- Decision:
  - implement as lists of int
- Problem:
  - lists have order and repetition, sets don't
- Solution:
  - use only lists of int ordered from smallest to largest with no repetition (data invariant)

Abstract Type – SML code Example

```
type intset = int list
val empty_set = []:intset
fun insert {elt, set = [] } = [elt]
      |  insert {elt, set = x :: xs} =
         if elt < x then elt :: x :: xs
         else if elt = x then x :: xs
         else x :: (insert {elt = elt, set = xs})

fun union ([],ys) = ys
    | union (x::xs,ys) =
       union(xs,insert{elt=x,set = ys})

fun intersect ([],ys) = []
   | intersect (xs,[]) = []
   | intersect (x::xs,y::ys) =
         if x <y then intersect(xs, y::ys)
         else if y < x then intersect(x::xs,ys)
         else x :: (intersect(xs,ys))
```

```
fun elt_of {elt, set = []} = false
   | elt_of {elt, set = x::xs} =
      (elt = x) orelse
      (elt > x andalso
       elt_of{elt = elt, set = xs})
```

Abstract Type – Example

- Notice that all these definitions maintain the data invariant for the representation of sets, and depend on it

- Are we happy now?
- NO!
- As is, user can create any pair of lists of int and apply union to them; the result is meaningless

Solution: abstract datatypes

```
abstype intset = Set of int list with
val empty_set = Set []
local
fun ins {elt, set = [] } = [elt]
   |  ins {elt, set = x :: xs} =
        if elt < x then elt :: x :: xs
        else if elt = x then x :: xs
        else x :: (ins {elt = elt, set =
         xs})
fun un ([],ys) = ys
   | un (x::xs,ys) =
      un (xs,ins{elt=x,set = ys})
in
      fun insert {elt, set = Set s}=
            Set(ins{elt = elt, set  = s})
      fun union (Set xs, Set ys) =
          Set(un (xs, ys))
end
```

```
local
fun inter ([],ys) = []
   | inter (xs,[]) = []
   | inter (x::xs,y::ys) =
        if x <y then inter(xs, y::ys)
        else if y < x then inter(x::xs,ys)
        else x :: (inter(xs,ys))

in
      fun intersect(Set xs, Set ys) =
          Set(inter(xs,ys))
end
fun elt_of {elt, set = Set []} = false
   | elt_of {elt, set = Set (x::xs)} =
      (elt = x) orelse
      (elt > x andalso
       elt_of{elt = elt, set = Set xs})
fun set_to_list (Set xs) = xs
end (* abstype *)
```

Abstract Type – Example

- Creates a new type (not equal to **int list**)
  - Remember type equivalence – structure vs. name
- Exports
  - type **intset,**
  - Constant **empty_set**
  - Operations: **insert, union, elt_of,** and **set_to_list**; act as primitive

  - Note: Unfortunately in SML we cannot use pattern matching or list functions on intset; won't type check
  - Lack of orthogonality in the design of abstype for SML – does not fulfill Tennent's principle of data type completion

Abstract Type – Example

- Implementation: just use **int list**, except for type checking
- Data constructor **Set** only visible inside the asbtype declaration; type intset visible outside
- Function **set_to_list** used only at compile time
- Data abstraction allows us to prove data invariant holds for all objects of type intset

Abstract Types

- A type is abstract if the user can only see:
  - the type
  - constants of that type (by name)
  - operations for interacting with objects of that type that have been explicitly exported
- Primitive types are built-in abstract types

  e.g. **int** type in Java
  - The representation is hidden
  - Operations are all built-in
  - User programs can define objects of **int** type
- User-defined abstract data types must have the same characteristics as built-in abstract data types

User Defined Abstract Types

- Syntactic construct to provide encapsulation of abstract type implementation
- Inside, implementation visible to constants and subprograms
- Outside, only type name, constants and operations visible, not implementation
- No runtime overhead as all the above can be checked statically

Advantages of Data Abstraction

- Advantage of Inside condition:
  - Program organization, modifiability (everything associated with a data structure is together)
  - Separate compilation may be possible

- Advantage of Outside condition:
  - Reliability--by hiding the data representations, user code cannot directly access objects of the type. User code cannot depend on the representation, allowing the representation to be changed without affecting user code.

Limitation of Abstract data types

Queue

```
abstype q
with
  mk_Queue : unit -> q
  is_empty : q -> bool
  insert   : q * elem -> q
  remove   : q -> elem
is  …
in
  program
end
```

Priority Queue

```
abstype pq
with
  mk_Queue : unit -> pq
  is_empty : pq -> bool
  insert   : pq * elem -> pq
  remove   : pq -> elem
is …
in
  program
end
```

But cannot intermix pq's and q's

Abstract Data Types

- Guarantee invariants of data structure
  - only functions of the data type have access to the internal representation of data
- Limited "reuse"
  - Cannot apply queue code to pqueue, except by explicit parameterization, even though signatures identical
  - Cannot form list of points and colored points

- Data abstraction is important – how can we make it extensible?
- Remember subtyping from Lecture 13 ?

Subtyping for Product Types

The rule is:

if $A <: T$ and $B <: U$ then $A \times B <: T \times U$

This rule, and corresponding rules for other structured types, can be worked out by following the principle:

$T <: U$ means that whenever a value of type U is expected, it is safe to use a value of type T instead.

What can we do with a value *v* of type $T \times U$?

- use *fst(v)* , which is a value of type T
- use *snd(v)* , which is a value of type U

If *w* is a value of type $A \times B$ then *fst(w)* has type A and can be used instead of *fst(v)*. Similarly *snd(w)* can be used instead of *snd(v)*. Therefore *w* can be used where *v* is expected.

Objects and subtyping

- Objects can be thought of as (extendible) records of fields and methods.
- That is why Square <: Shape and Circle <: Shape in

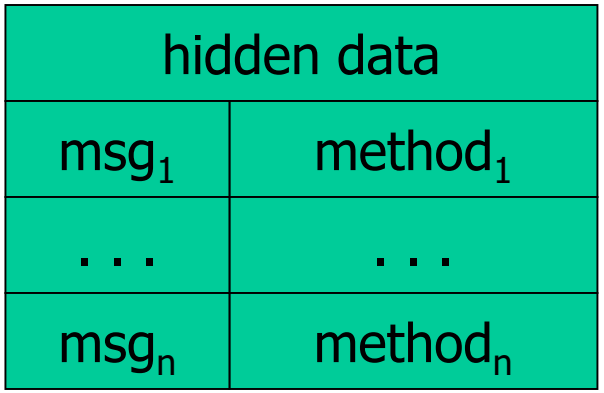
```
abstract class Shape {
  abstract float area( ); }
```

```
class Square extends Shape {
  float side;
  float area( ) {return (side * side); } }
```

```
class Circle extends Shape {
  float radius;
  float area( ) {return ( PI * radius * radius); } }
```

Objects

- An object consists of
  - hidden data
    - instance variables, also called member data
    - hidden functions also possible
  - public operations
    - methods or member functions
    - can also have public variables in some languages

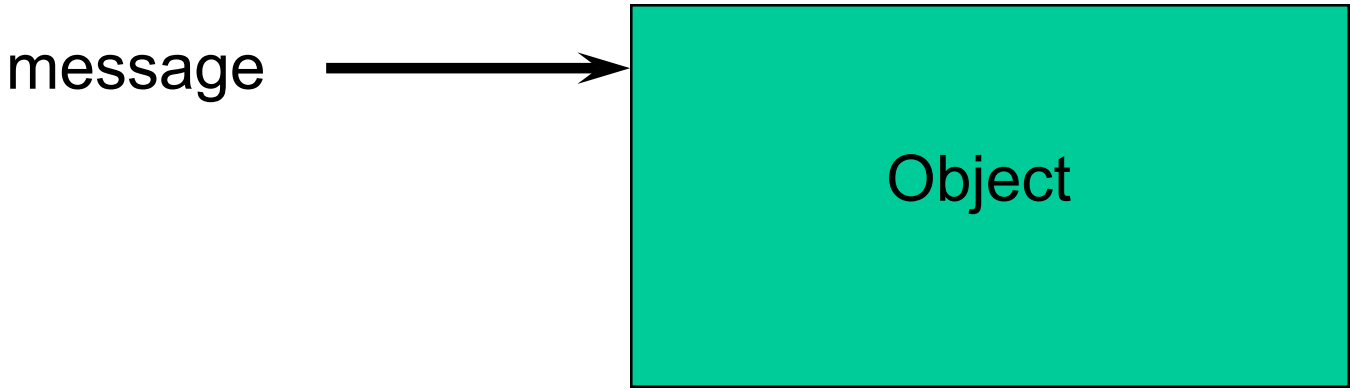


Objects can be extended by cloning or subclassing

- Object-oriented program:
  - Send messages to objects:
    - o → m (a) or o.m(a)

Encapsulation

- Builder of a concept has detailed view
- User of a concept has “abstract” view
- Encapsulation is the mechanism for separating these two views
- The message concept facilitate loose coupling



Object-oriented programming

- Metaphor usefully ambiguous
  - Database, window, file, integer – all are objects
  - sequential or concurrent computation
  - distributed, sync. or async. Communication
- Programming methodology
  - organize concepts into objects and classes
  - build extensible systems
- Language concepts
  - encapsulate data and functions into objects
  - subtyping allows extensions of data types
  - inheritance allows reuse of implementation
  - dynamic lookup facilitate loose coupling

Dynamic Lookup – dynamic dispatch

- In object-oriented programming,

  object → message (arguments)

  object.method(arguments)

  code depends on object and message

  – Add two numbers x → add (y) or x.add(y)

    different add if x is integer or complex

- In conventional programming,

  operation (operands)

  meaning of operation is always the same

  – Conventional programming add (x, y)

    function add has fixed meaning

Dynamic Dispatch Example

```
class point {
    int c;
    int getColor() { return(c); }
    int distance() { return(0); }
}
class cartesianPoint extends point{
    int x, y;
    int distance() { return(x*x + y*y); }
}
class polarPoint extends point {
    int r, t;
    int distance() { return(r*r); }
    int angle() { return(t); }
}
```

Dynamic Dispatch Example

```
if (x == 0) {
    p = new point();
} else if (x < 0) {
    p = new cartesianPoint();
} else if (x > 0) {
    p = new polarPoint();
}
y = p.distance();
```

Which distance method is invoked?

- Invoked Method Depends on Type of Receiver!
  - if p is a point
    - return(0)
  - if p is a cartesianPoint
    - return(x*x + y*y)
  - if p is a polarPoint
    - return(r*r)

Dynamic dispatch

- If methods are overridden, and if the PL allows a variable of a particular class to refer to an object of a subclass, then method calls entail **dynamic dispatch**.

- Consider the Java method call $O.M(E_1, \ldots, E_n)$:
  - The compiler infers the type of *O*, say class *C*.
  - The compiler checks that class *C* is equipped with a method named *M*, of the appropriate type.
  - Nevertheless, it might turn out (at run-time) that the target object is actually of class *S*, a subclass of *C*.
  - If method *M* is overridden by any subclass of *C*, a run-time tag test is needed to determine the actual class of the target object, and hence which of the methods named *M* is to be called.

Overloading vs. Dynamic Dispatch

- Dynamic Dispatch
  - Add two numbers x.add (y)

    different add if x is integer, complex, ie. depends on the run-time type of x

- Overloading
  - add (x, y) function add has fixed meaning
  - int-add if x and y are ints, i.e. add (int x, int y)
  - real-add if x and y are reals i.e. add (float x, float y)

Important distinction:

Overloading is resolved at compile time,

Dynamic lookup at run time.

Comparison

- Traditional approach to encapsulation is through abstract data types
- Advantage
  - Separate interface from implementation
- Disadvantage
  - All ADTs are independent and at the same level
  - Not extensible in the way that OOP is
  - Not reusable in the way OOP is

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

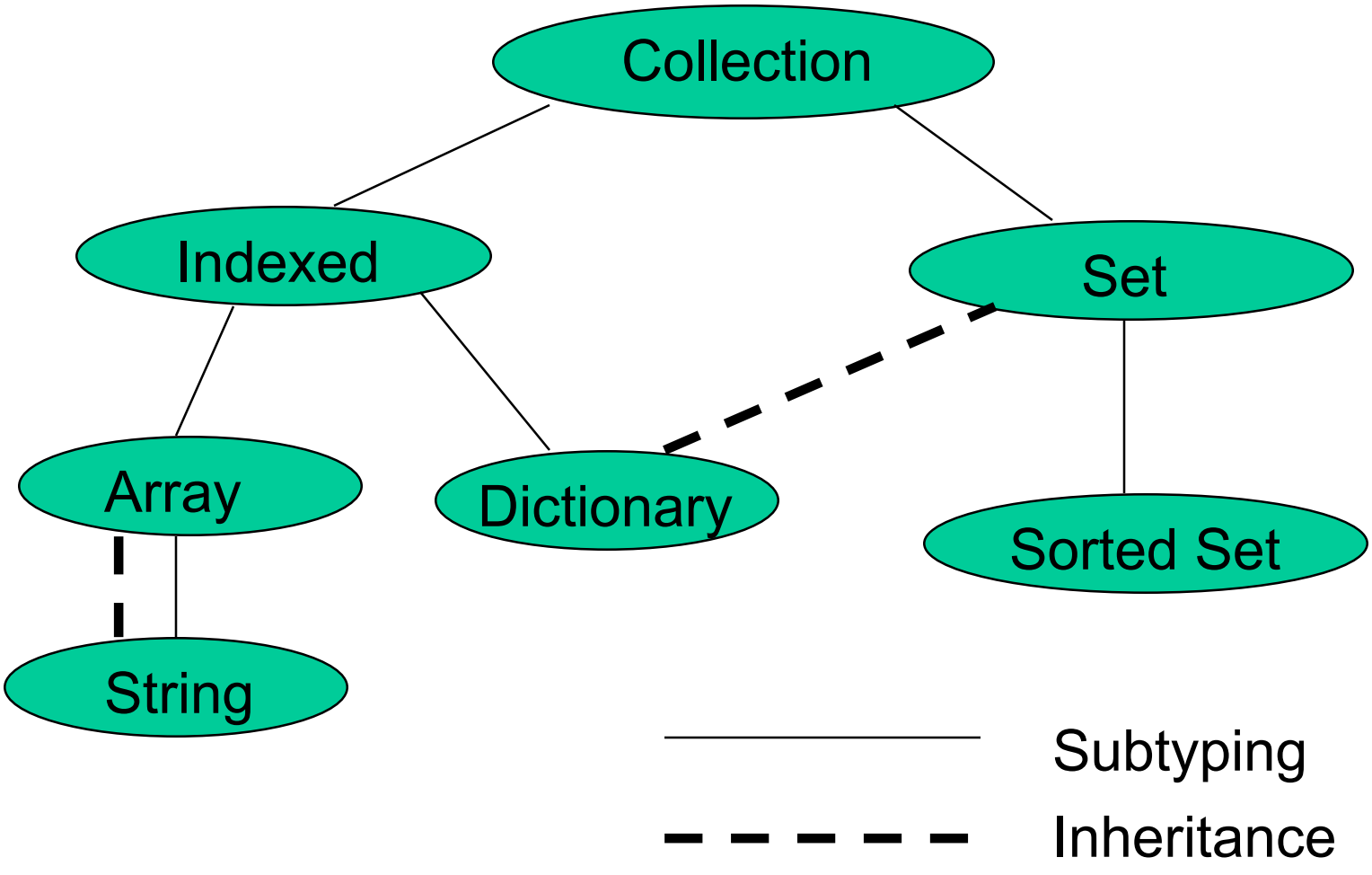
- Implementation mechanism
- New objects may be defined by reusing implementations of other objects

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
  - Property used by implementor of classes

Subtyping differs from inheritance



Inheritance

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

- Note in Java and C# inheritance also implies a subtype relation !

- In C++ you can have inheritance without subtyping by extending a class private:
  - class Derived: private Base { … };
  -

Tennent's Language Design principles and OOP

- The Principle of Abstraction
  - All major syntactic categories should have abstractions defined over them. For example, functions are abstractions over expressions

- We have seen abstractions over expressions, i.e. functions
- We have seen abstractions over commands, i.e. procedures
- What about abstractions over declarations?
  - Well Tennent, in 1981 saw that
  - Declabs Name(params) begin D end
  - Is exactly the notion of a class in the simula language !
  - "but this is not a widespread language construct"
  - Well not in 1981 ☺

Varieties of OO languages

- class-based languages
  - behaviour of object determined by its class
- object-based
  - objects defined directly
- multi-methods
  - operation depends on all operands

History

- Simula 1960's
  - Object concept used in simulation
- Smalltalk 1970's
  - Object-oriented design, systems
- C++ 1980's
  - Adapted Simula ideas to C
- Java 1990's
  - Distributed programming, internet
- C# 2000's
  - Combine the efficiency of C/C++ with the safety of Java
- Scala,F#, Swift, RUST - combine FP and OOP 2010's

Runtime Organization for OO Languages

How to represent/implement object oriented constructs such as **objects**, **classes, methods**, **instance variables** and **method invocation**

Some definitions for these concepts:

- An **object** is a group of instance variables to which a group of instance methods is attached.
- An **instance variable** is a named component of a particular object.
- An **instance method** is a named operation attached to a particular object and able to access that objects instance variables
- An **object class** (or just **class)** is a family of objects with similar instance variables and identical methods.

Runtime Organization for OO Languages

Objects are a lot like records, and instance variables are a lot like fields.
=> The representation of objects is similar to that of a record.

Methods are a lot like procedures.
=> Implementation of methods is similar to routines.

But… there are differences:

Objects have methods as well as instance variables, records only have fields (except in C#).

The methods have to somehow know what object they are associated with (so that methods can access the object's instance variables)

Example

A simple Java object (no inheritance)

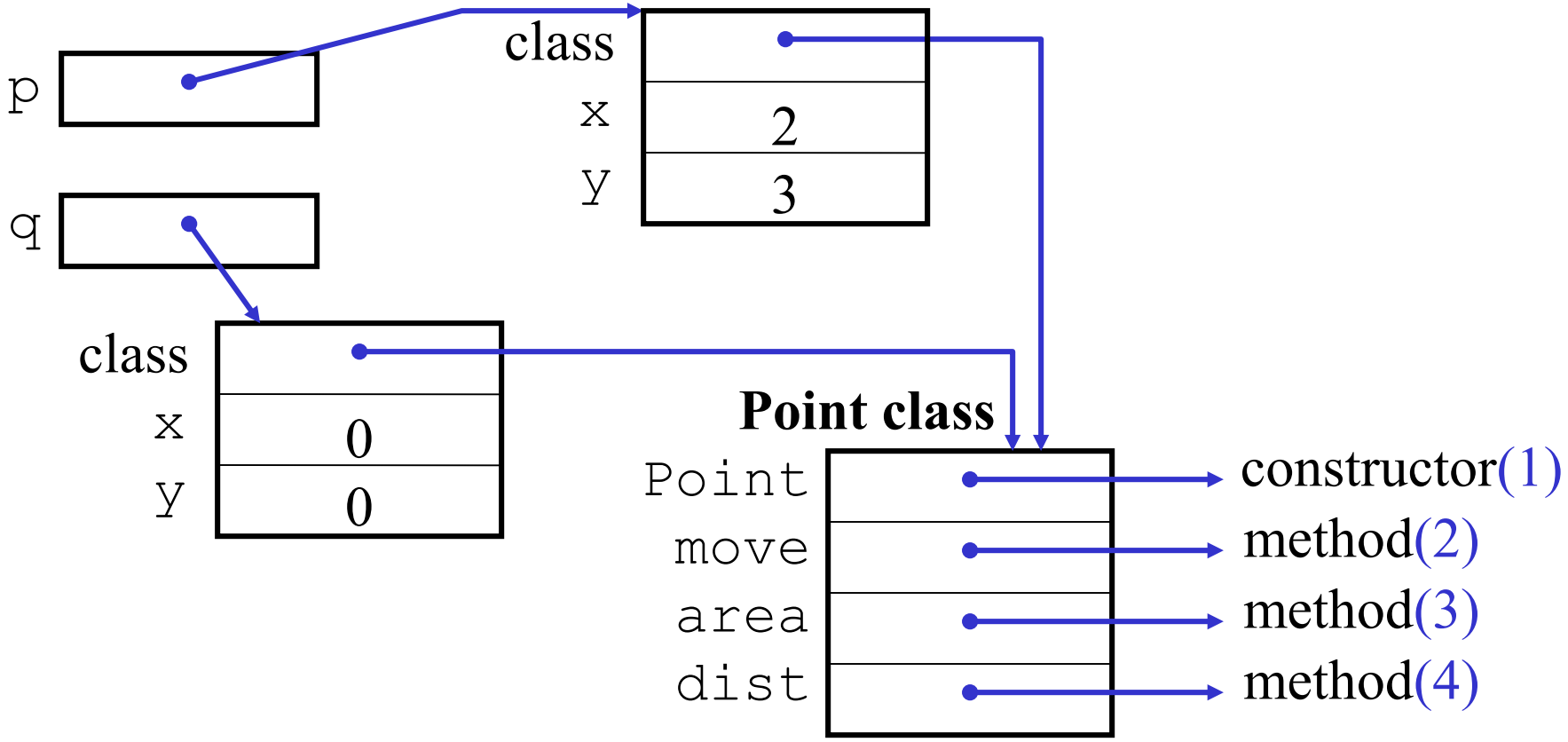
```
class Point {
  int x,y;
(1)public Point(int x, int y) {
     this.x=x; this.y=y;
  }

(2)public void move(int dx, int dy) {
     x=x+dx; y=y+dy;
  }

(3)public float area() { ...}
(4)public float dist(Point other) { ... }
}
```

Example

Representation of a simple Java object (no inheritance)

```
Point p = new Point(2,3);
Point q = new Point(0,0);
```

new allocates an object in the heap



Example

Points and other “shapes” (Inheritance)

```
abstract class Shape {
  int x,y; // “origin” of the shape
(S1) public Shape(int x, int y) {
      this.x=x; this.y=y;
  }

(S2) public void move(int dx, int dy) {
      x=x+dx; y=y+dy;
  }

  public abstract float area();
(S3) public float dist(Shape other) { ... }
}
```

Example

Points and other “shapes” (Inheritance)

```
class Point extends Shape {

(P1) public Point(int x, int y) {
        super(x,y);
     }

(P2) public float area() { return 0.0; }
}
```

Example

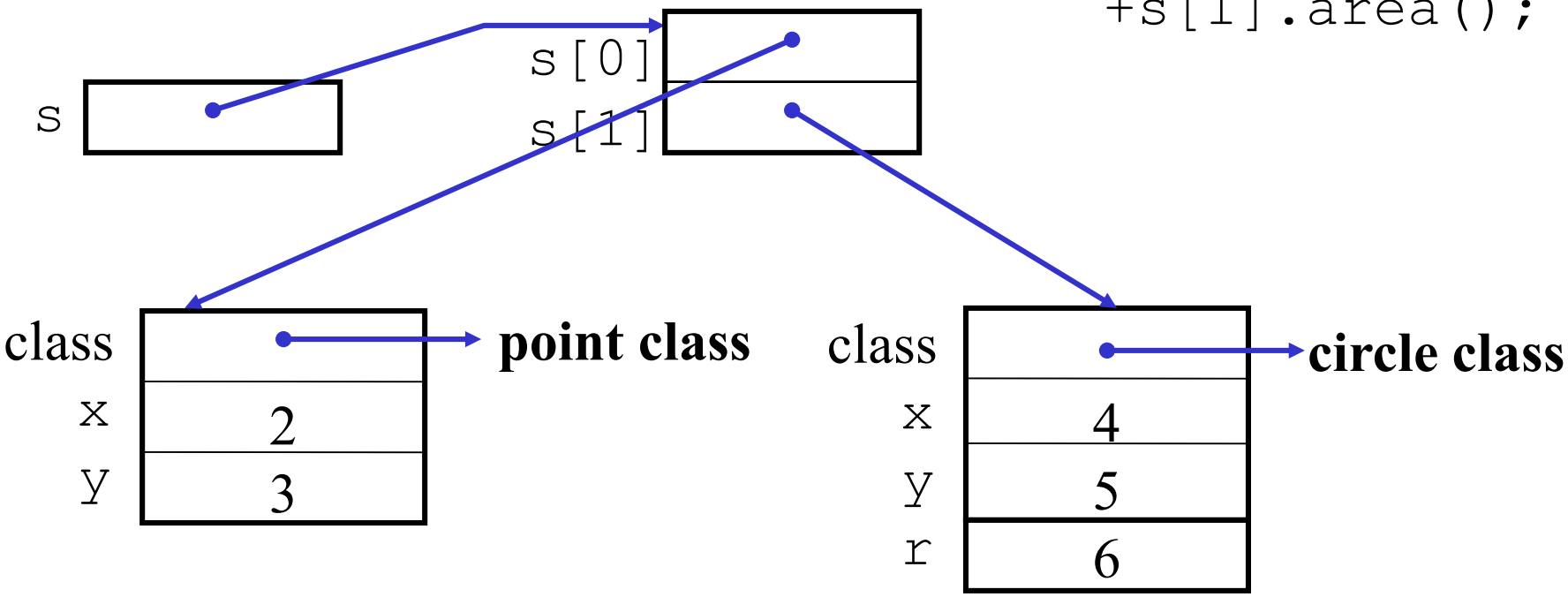
Points and other “shapes” (Inheritance)

```
class Circle extends Shape {
  int r;
(C1)public Circle(int x,int y,int r) {
      super(x,y); this.r = r;
  }

(C2)public int radius() { return r; }

(C3)public float area() {
      return 3.14 * r * r;
  }
}
```

Representation of Points and other "shapes" (Inheritance)

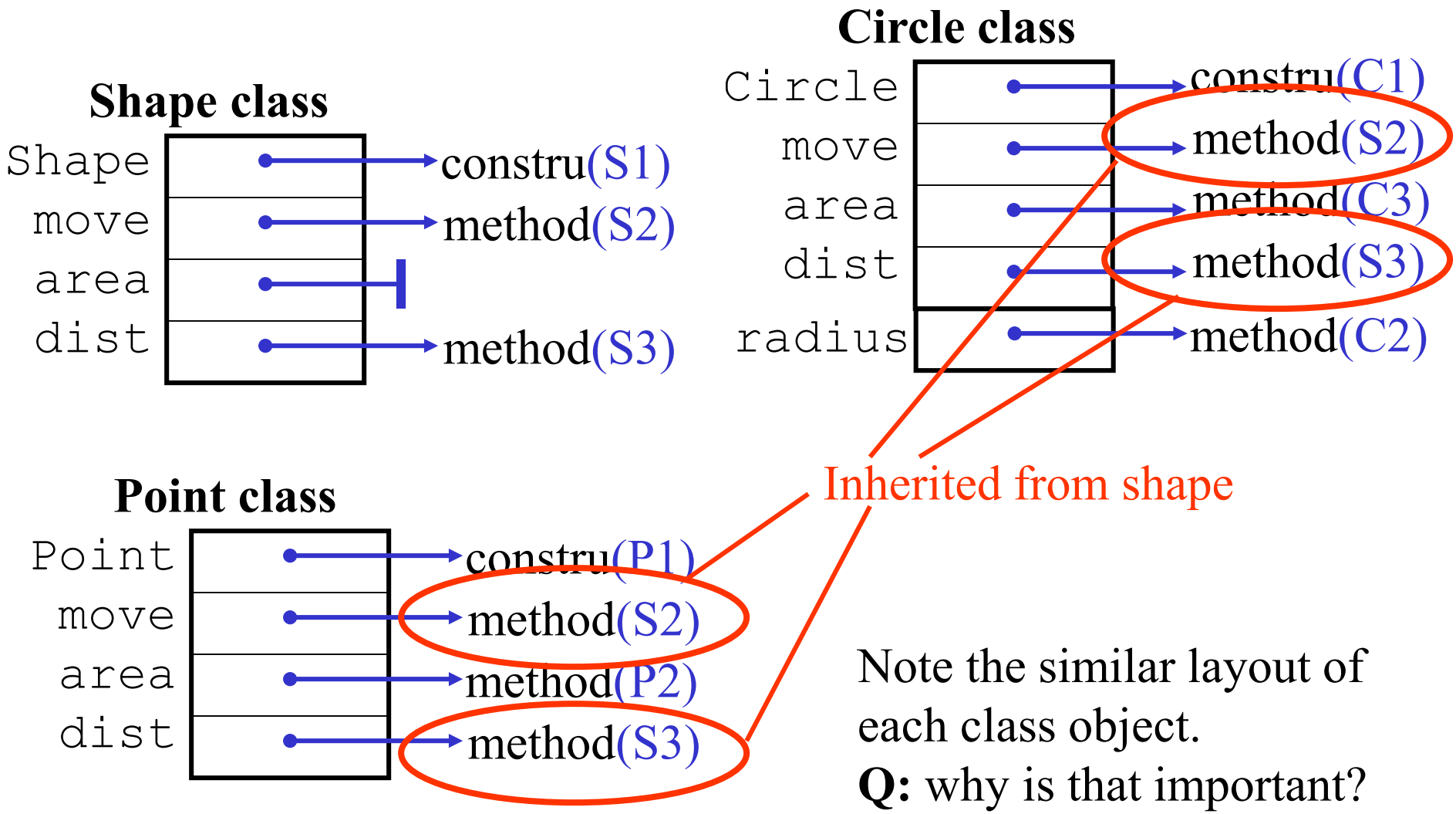
```
Shape[] s = new Shape[2];
s[0] = new Point(2,3);
s[1] = new Circle(4,5,6);
```

```
s[0].x = ...;
s[1].y = ...;
float areas =
    s[0].area()
   +s[1].area();
```



Note the similar layout between point and circle objects!

Representation of Points and other "shapes" (Inheritance)

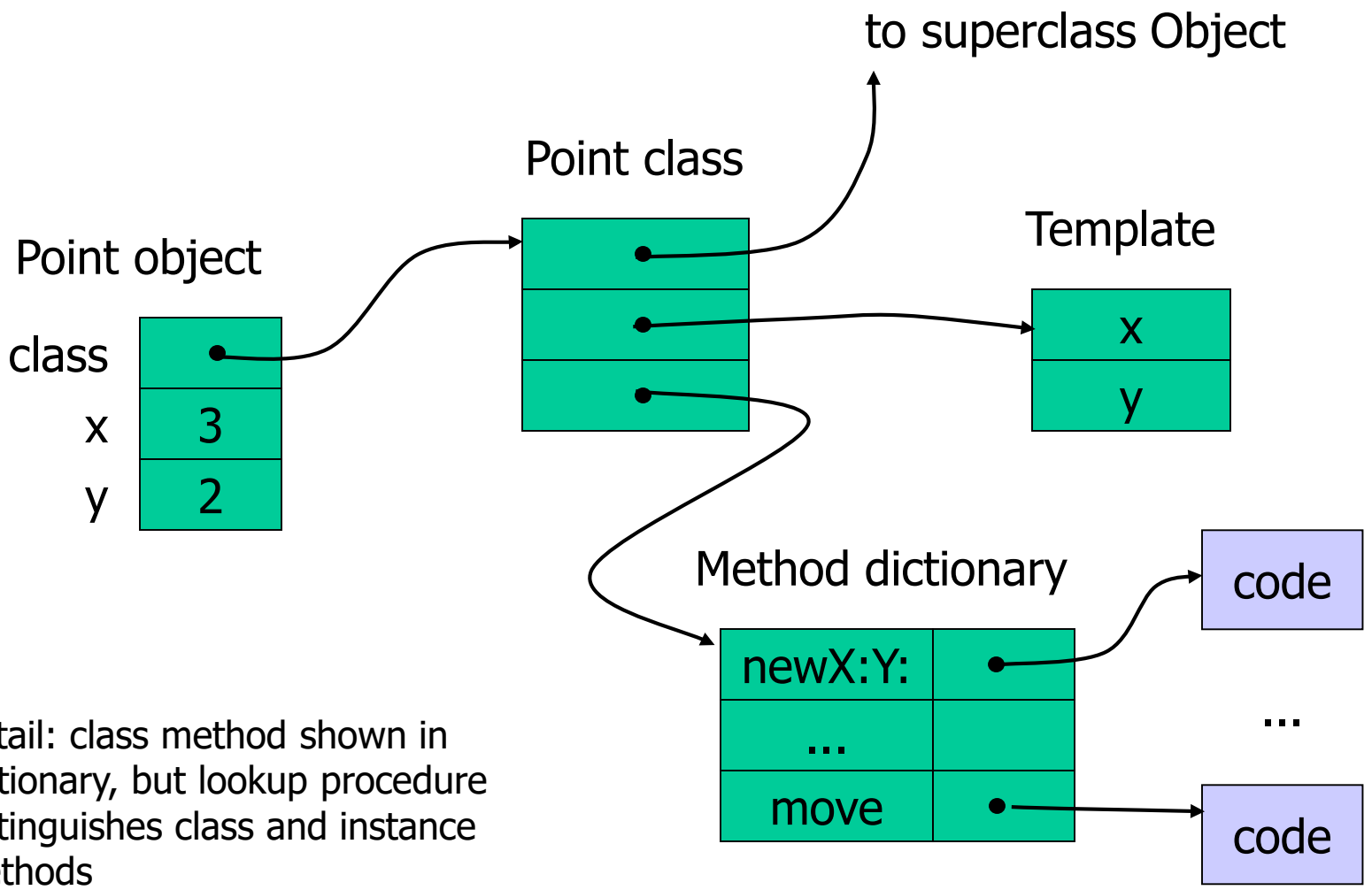


Note the similar layout of each class object.

Q: why is that important?

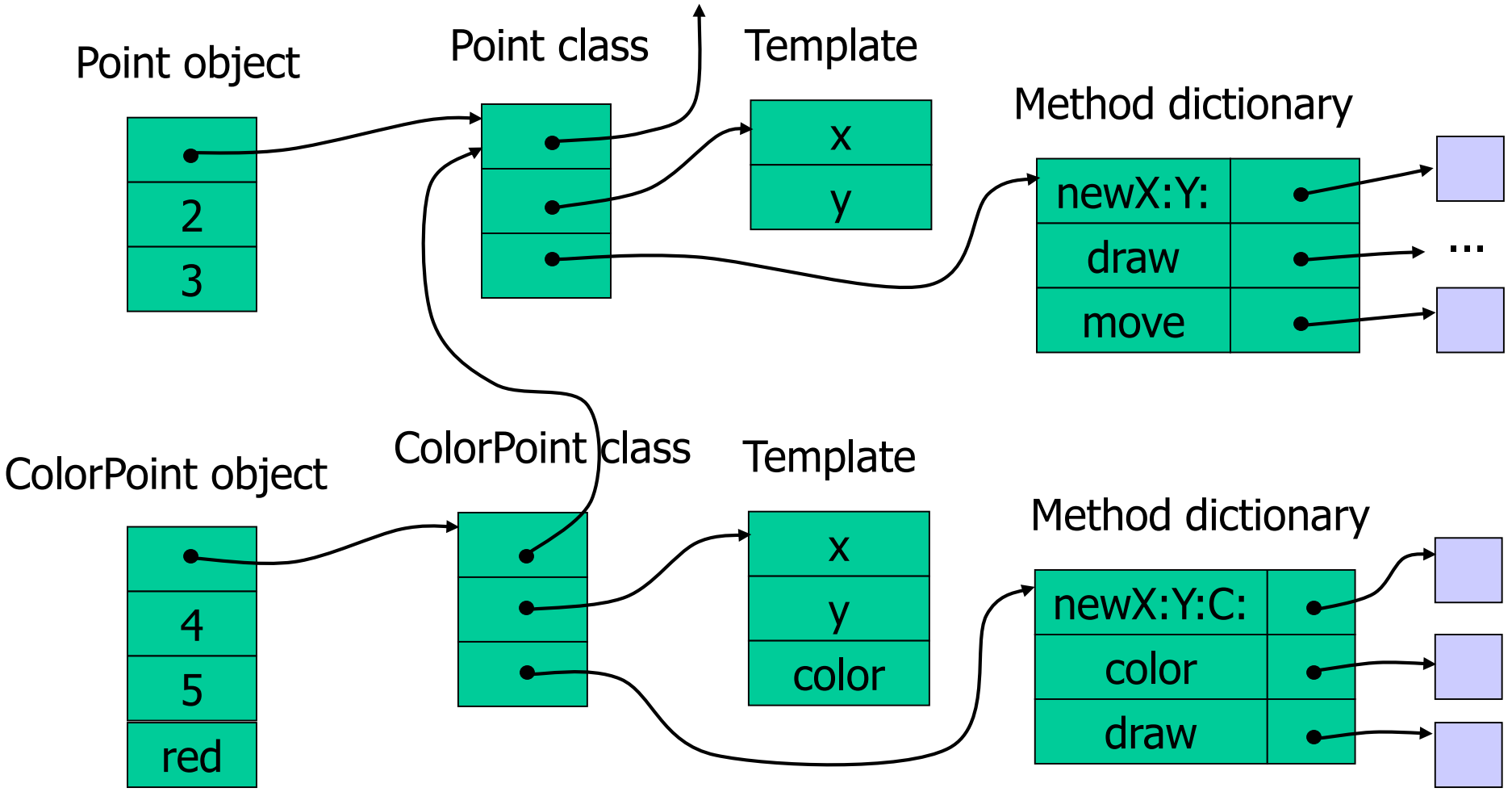
Q: why don't we need a pointer to the super class in a class object?

Alternative Run-time representation of point



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

Alternative Run-time representation



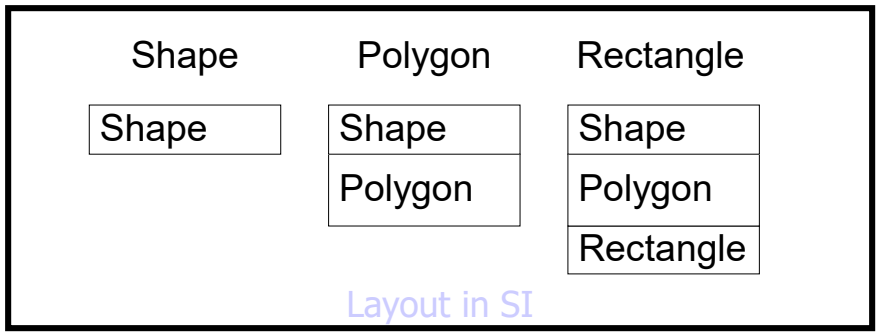
This is a schematic diagram meant to illustrate the main idea. Actual implementations may differ.

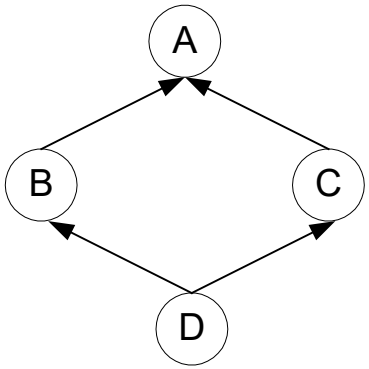
Multiple Inheritance

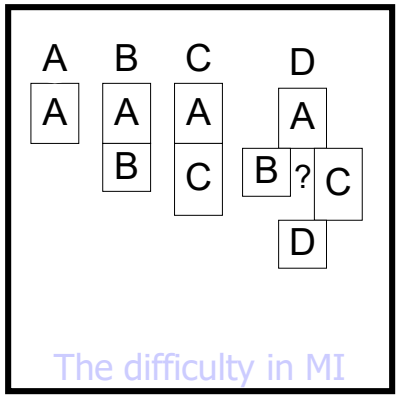
- In the case of single inheritance, each class may have one direct predecessor; multiple inheritance allows a class to have several direct predecessors.
- In this case the simple ways of accessing attributes and binding method-calls (shown previously) don't work.
- The problem: if class C inherits class A and class B the objects of class C cannot begin with attributes inherited from A and at the same time begin with attributes inherited from B.
- In addition to these implementation problems multiple inheritance also introduces problems at the language (conceptual) level.

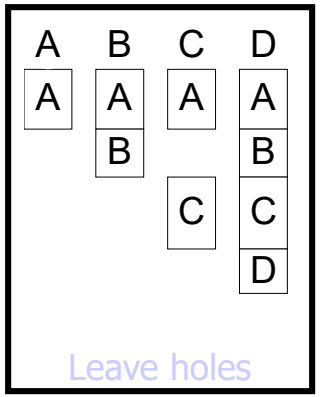
Object Layout

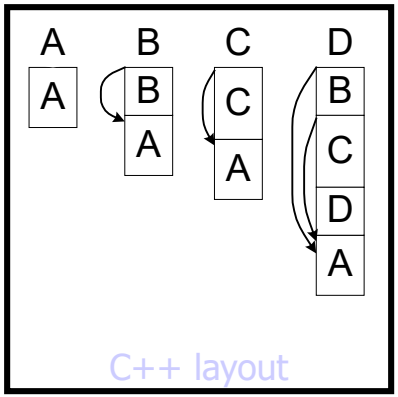
- The memory layout of the **object's fields**
- How to access a field if the dynamic type is unknown?
  - Layout of a type must be "compatible" with that of its supertypes
  - Easy for Single Inheritance hierarchies
    - The new fields are added at the end of the layout
  - Hard for MI hierarchies

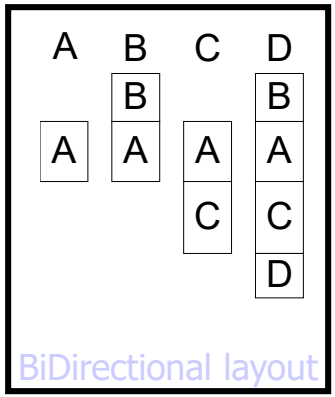












Dynamic (late) Binding

- Consider the method call:
  - x.f(a,b) where is f defined?
    in the class (type)of x? Or in a predecessor?
- If multiple inheritance is supported then the entire predecessor graph must be searched:
  - This costs a large overhead in dynamic typed languages like Smalltalk (normally these languages don't support multiple inheritance)
  - In static typed languages like Java, Eiffel, C++ the compiler is able to analyse the class-hierarchy (or more precise: the graph) for x and create a display-array containing addresses for all methods of an object (including inherited methods)
  - According to Meyer the overhead of this compared to static binding is at most 30%, and overhead decreases with complexity of the method
- If multi-methods are supported a forest like data structure has to be searched

Traits

- Some feel that single inheritance is too limiting
- Interface specification helps by forcing class to implement specified methods, but can lead to code duplication
- A trait is a collection of *pure* methods
- Can be thought of as an interface with implementation
- Classes “use” traits
- Traits can be used to supply the same methods to multiple classes in the inheritance hierarchy

Simple Example Using Traits

```
trait Similarity {
  def isSimilar(x: Any): Boolean
  def isNotSimilar(x: Any): Boolean = !isSimilar(x)
}
```

- This trait consists of two methods isSimilar and isNotSimilar
  - isSimilar is abstract
  - isNotSimilar is concrete but written in terms of isSimilar
- Classes that integrate this trait only have to provide a concrete implementation for isSimilar, isNotSimilar gets inherited directly from the trait

Simple Example Using Traits

```
class Point(xc: Int, yc: Int) extends
Similarity {
var x: Int = xc
var y: Int = yc
  def isSimilar(obj: Any) =
      obj.isInstanceOf[Point] &&
      obj.asInstanceOf[Point].x == x
}
```

Using Traits

- *Class = Superclass + State + Traits + Glue*
- A class provides it's own state
- It also provides "glue", which is the code that hooks the traits in
- Traits can satisfy each other's requirements for accessors
- A class is *complete* if all of the trait's requirements are met

- Languages with traits:
  - SmallTalk/Squeak/Pharo
  - Fortress
  - Scala
  - Swift
  - Kotlin
  - (Java8 – default methods on interfaces)

Implementation of Object Oriented Languages

- Implementation of Object Oriented Languages differs only slightly from implementations of block structured imperative languages
- Some additional work to do for the contextual analysis
  - Access control, e.g. private, public, protected directives
  - Subtyping can be tricky to implement correctly
- The main difference is that methods usually have to be looked up dynamically, thus adding a bit of run-time overhead
  - For efficiency some languages introduce modifiers like:
    - **final** (Java) or **virtual/override** (C#)
  - Multiple inheritance poses a bigger problem
  - Multi methods pose an even bigger problem

Larger Encapsulation Constructs

- Original motivation:
  - Large programs have two special needs:
  1. Some means of organization, other than simply division into subprograms
  2. Some means of partial compilation (compilation units that are smaller than the whole program)
- Obvious solution: a grouping of subprograms that are logically related into a unit that can be separately compiled (compilation units)
  - These are called encapsulations (or packages or modules)
  - Classes are too small (unless they allow true inner classes)

Encapsulation Constructs

- Why are Classes are too small and what is true inner classes?
- Originally mainstream OOP languages like C++ and Java had a flat namespace for classes
- But what if classes can be declared within classes?
- Java 1.1 introduced the notion of inner/nested classes:

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

```
class OuterClass {
    ...
    static class StaticNestedClass {
        ...
    }
}
```

- Distinction between inner and nested classes
  - An inner class refer to an instance of the outer class in Java
  - A nested class is declared as static in Java
  - C# and C++ have static nested classes
    - remember in C# members are static unless declared to be overridable and inner classes cannot be declared overridable

Encapsulation Constructs

- Static nested classes introduce a form of namespace hierachy:

  OuterClass.StaticNestedClass nestedObject =

  new OuterClass.StaticNestedClass();

  Static nested classes only have access to static members and methods!

- Inner classes needs an instance of the outer class:

  OuterClass outerObject = new OuterClass();

  OuterClass.InnerClass innerObject =

  outerObject.new InnerClass();

  Inner classes have access to members and methods of the instance of the outer class

  Note Java also allow class definitions in methods, but

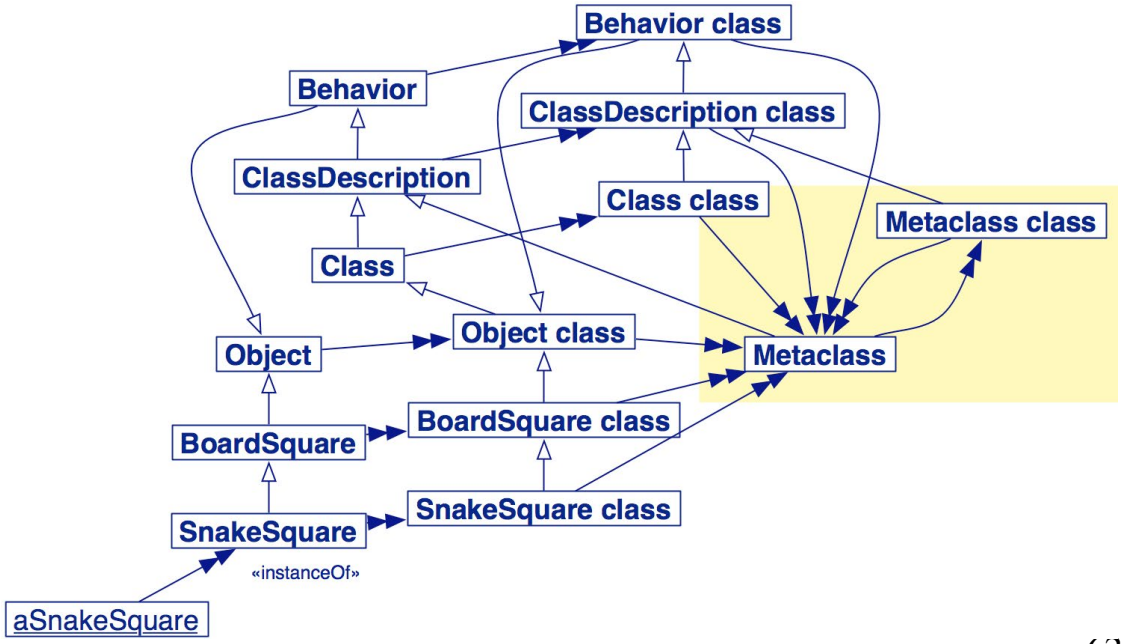
Encapsulation Constructs

- However, many restrictions on inner classes in Java
  - A method can declare a local class,
    - but only access to variables declared as final – a restriction put to ensure a closure is not needed.
  - Classes cannot be treated as objects in Java.
- Other languages treat classes as first class objects
  - E.g. SmallTalk:

    Every object has

    a class and every

    Class is an object



Naming Encapsulations

- Large programs define many global names
- So we need a way to divide names into logical groupings
- A naming encapsulation is used to create a new scope for names
- C++ Namespaces
  - Can place each library in its own namespace and qualify names used outside with the namespace
- C# also includes namespaces
- In Java namespaces are called packages

Naming Encapsulations

- Java Packages
  - Packages can contain more than one class definition; classes in a package are partial friends
  - Clients of a package can use fully qualified name or use the `import` declaration
- Ada Packages
  - Packages are defined in hierarchies which correspond to file hierarchies
  - Visibility from a program unit is gained with the `with` clause
- SML Modules
  - Called **structure**; interface called **signature**
  - Interface specifies what is exported
  - Interface and structure may have different names
  - If structure has no signature, everything exported
  - Modules may be parameterized (**functors)**
  - Module system quite expressive

Modules

- Language construct for grouping related types, data structures, and operations
- Typically allows at least some encapsulation
  - Can be used to provide abstract types
- Provides scope for variable and subprogram names
- Typically includes interface stating which modules it depends upon and what types and operations it exports
- Compilation unit for separate compilation

Encapsulation Constructs

- Encapsulation in C
  - Files containing one or more subprograms can be independently compiled
  - The interface is placed in a header file (.h)
  - Problem: the linker does not check types between a header and associated implementation
- Encapsulation in C++
  - Similar to C
  - Addition of friend functions that have access to private members of the friend class

Encapsulation Constructs

- Ada Package
  - Can include any number of data and subprogram declarations
  - Two parts: specification and body
  - Can be compiled separately
- C# Assembly
  - Collection of files that appears to be a single dynamic link library or executable
  - Larger construct than class; used by all .NET programming languages
- Java Module System (JSR 277/JSR376)
  - New deployment and distribution format
  - New language constructs:
    - module, import/export, provides/requires

Java 9 module system


```
module X {
  requires Y;
}
```

```
module Y {
  exports Q;
}
```

Issues for modules

- The target language usually has one name space
  - Generate unique names for modules
  - Some assemblers support local names per file
  - Use special characters which are invalid in the programming language to guarantee uniqueness
    - This is what Java does since the JVM has no nested classes

- Generate code for initialization
  - Modules may use items from other modules
  - Init before used
  - Init only once
  - Circular dependencies
    - How to initialize C once if module A uses module B and C, and B uses C
      - Compute a total order and init before use
      - Use special compile-time flag

Summary

- Abstract Data Types
  - Encapsulation
  - Invariants may be preserved
- Objects
  - Reuse
  - Subtyping
  - Inheritance
  - Dynamic dispatch
- Modules
  - Grouping (related) entities
  - Namespace management
  - Separate compilation

“I invented the term *Object-Oriented* and I can tell you I did not have C++ in mind.”

Alan Kay

Inventor of Smalltalk