COMP 412
FALL 2010

The Procedure Abstraction, Part V: Support for OOLs

Comp 412

What about Object-Oriented Languages?

What is an OOL?

- A language that supports "object-oriented programming"

How does an OOL differ from an ALL? (ALGOL-Like Language)

- Data-centric name scopes for values & functions
- Dynamic resolution of names to their implementations

How do we compile OOLs ?

- Need to define what we mean by an OOL
- Term is almost meaningless today —
  - Smalltalk to C++ to Java
- We will focus on Java and C++
- Differences from an ALL lie in naming and addressability

OOL ≅ Object-Oriented Language
ALL ≅ Algol-Like Language

What Are The Issues?

In an ALL, the compiler needs

- Compile-time mechanism for name resolution
- Runtime mechanism to compute an address from a name

Compiler must emit code that builds & maintains the runtime structures for addressability

In an OOL, the compiler needs

- Compile-time mechanism for name resolution
- Runtime mechanism to compute an address from a name

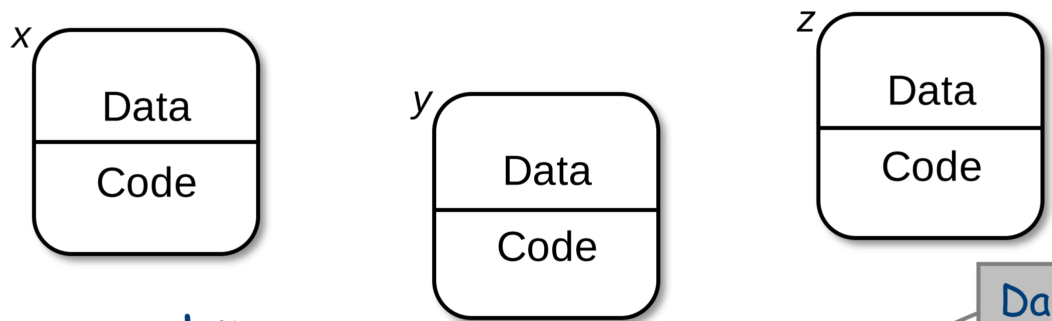
Compiler must emit code that builds & maintains the runtime structures for addressability

This lecture focuses on these three mechanisms.
The issue of code generation for a method call is discussed in a later lecture.

What is an Object?

An object is an abstract data type that encapsulates data, operations and internal state behind a simple, consistent interface.

The Concept:



Elaborating the concepts:

- Each object has internal state
  - Data members are static (*lifetime of object*)
  - External access is through code members
- Each object has a set of associated procedures, or methods
  - Some methods are public, others are private
  - Locating a procedure by name is more complex than in an ALL
- Object's internal state leads to complex behavior

Data members, variables

Code members, or methods

OOLs & the Procedure Abstraction

What is the shape of an OOL's name space?

- Local storage in objects (*both public & private*)
- Storage defined in methods (*they are procedures*)
  - Local values inside a method
  - Static values with lifetimes beyond methods
- Methods shared among multiple objects
- Global name space for global objects and (*some?*) code

In some OOLs, everything is an object.

In others, variables co-exist with objects & inside objects.

Classes

- Objects with the same ~~state~~ members are grouped into a *class*
  - Same code, same data, same naming environment
  - Class members are static & shared among instances of the class
- Allows abstraction-oriented naming
- Should foster code reuse in both source & implementation

Implementing Object-Oriented Languages

So, what can an executing method access?

The fundamental question

- Names defined by the method
  - *And its surrounding lexical context*
- The receiving object's data members
  - Smalltalk terminology: *instance variables*
- The code & data members of the class that defines it
  - *And its context from inheritance*
  - Smalltalk terminology: *class variables and methods*
- Any object defined in the global name space

Inheritance adds some twists.

The method might need the address for any of these objects

An OOL resembles an ALL, with a wildly different name space

- Scoping is relative to hierarchy in the code of an ALL
- Scoping is relative to hierarchy in the data of an ALL

Concrete Example: The Java Name Space

Code within a method M for object O of class C can see:

- Local variables declared within M (*lexical scoping*)
- All instance variables & class variables of C
- All public and protected variables of any <u>*superclass*</u> of C
- Classes defined in the same package as C or in any explicitly imported package
  - public class variables and public instance variables of imported classes
  - package class and instance variables in the package containing C
- Class declarations can be nested!
  - These member declarations hide outer class declarations of the same name (*lexical scoping*)
  - Accessibility: public, private, protected, package

class hierarchy

lexical

Both lexical nesting & class hierarchy at play

Superclass is an ancestor in the inheritance hierarchy

The Java Name Space

We will use and extend this example

```
Class Point {
    public int x, y;
    public void draw();
}
Class ColorPoint extends Point {              // inherits x, y, & draw() from Point
    Color c;                                  // local data
    public void draw() {…}                    // override (hide) Point's draw
    public void test() { y = x; draw(); }     //  local code
}
Class C {                                     // independent of Point & ColorPoint
    int x, y;                                 // local data
    public void m()                           // local code
    {
        Point p = new ColorPoint();           // uses ColorPoint, and, by inheritance
        y = p.x;                              // the definitions from Point
        p.draw();
    }
}
```

Java Symbol Tables

To compile method M of object O in class C, the compiler needs:

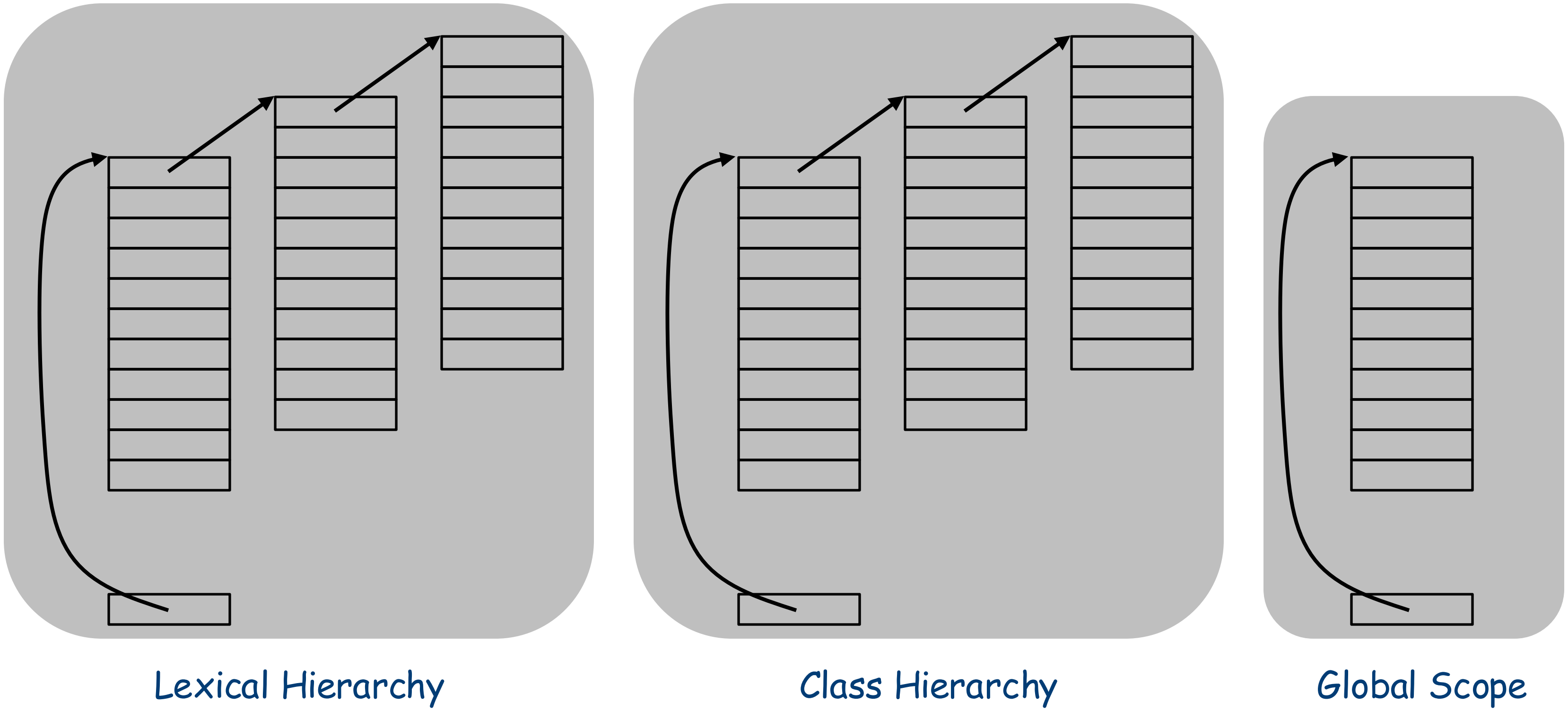
- Lexically scoped symbol table for the current block and its surrounding scopes
  - Just like ALL — inner declarations hide outer declarations
- Chain of symbol tables for inheritance
  - Class C and all of its superclasses
  - Need to find methods and instance variables in any superclass
- Symbol tables for all global classes (package scope)
  - Entries for all members with visibility
  - Need to construct symbol tables for imported packages and link them into the structure in appropriate places

Three sets of tables for name resolution

In an ALL, we could combine 1 & 3 for a single unified set of tables.

OOL Symbol Tables

Conceptually



Search Order: lexical, class, global

Java Symbol Tables

To find the address for a reference to *x* in method *M* for an object *O* of class *C*, the compiler must:

- For an unqualified use (i.e., x):
  - Search the symbol table for the method's lexical hierarchy
  - Search the symbol tables for the receiver's class hierarchy
  - Search global symbol table (current package and imported)
  - In each case check visibility attribute of x
- For a qualified use (i.e.: Q.x):
  - Find Q by the method above
  - Search from Q for x
    - → Must be a class or instance variable of Q or some class it extends
  - Check visibility attribute of x

Again, the "sheaf of tables" implementation makes simplifies the conceptual picture.

What Are The Issues?

In an ALL, the compiler needs

- Compile-time mechanism for name resolution
- Runtime mechanism to compute an address from a name

Compiler must emit code that builds & maintains the runtime structures for addressability

In an OOL, the compiler needs

- Compile-time mechanism for name resolution ✓
- Runtime mechanism to compute an address from a name ←

Compiler must emit code that builds & maintains the runtime structures for addressability

We need both a representation for each object and a mechanism to establish addressability of each object and its various members

Runtime Structures for OOLs

Object lifetimes are independent

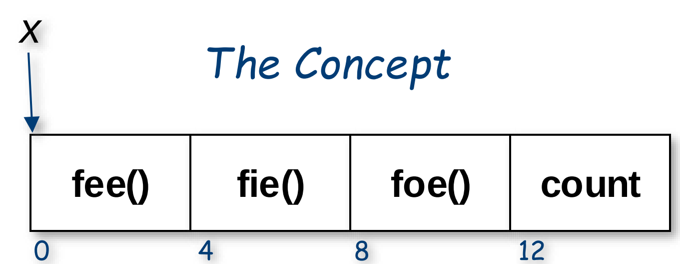
of method lifetimes,
of lifetimes of other objects …

- Each object needs an object record (OR) to hold its state
  - Independent allocation and deallocation
- Classes are objects, too
  - ORs of classes instantiate the class hierarchy

Object Records

- Static private storage for members
- Need fast, consistent access
  - Known constant offsets from OR pointer
- Provision for initialization

The Concept



Object Record Layout

Assume a Fixed-size OR

- Data members are at known fixed offsets from OR pointer
- Code members occur only in objects of class "class"
  - Code vector is a data-member of the class
  - Method pointers are at known fixed offsets in the code vector
  - Method-local storage kept in method's AR, as in an ALL
- Variable-sized members ⇒ store descriptor to space in heap

Locating ORs

- For a receiver, the OR pointer is implicit
- For a receiver's class, the receiver's OR has a class pointer
- Top-level classes and static classes can be accessed by name
  - Mangle the class name & use it as a relocatable symbol
  - Handle nested classes as we would nested blocks in an ALL

What About Inheritance?

Impact on OR Layout

- OR needs slots for each member declared, all the way up the class hierarchy (class, superclass, super-superclass, …)
- Can use prefixing of storage to lay out the OR

Back to Our Java Example — Class Point

```
Class Point {
    public int x, y;
    …
}
```

self →

class
x
y

OR for a Point

```
Class ColorPoint extends Point {
    Color c;
    …
}
```

self →

class
x
y
c

OR for a ColorPoint

What happens if we cast a ColorPoint to a Point?

Take the word <u>extends</u> literally.

Open World versus Closed World

Prefixing assumes that the class structure is known when layout is performed. Two common cases occur.

Closed-World Assumption (Compile time)

- Class structure is known and closed prior to runtime
- Can lay out ORs in the compiler and/or the linker

Open-World Assumption (Interpreter or JIT)

- Class structure can change at runtime
- Cannot lay out ORs until they are allocated
  - Walk class hierarchy at allocation

C++ has a closed class structure.

Java as an open class structure.