Runtime Support for OOLs
Object Records, Code Vectors, Inheritance
Comp 412

Copyright 2017, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapter 6 & 7 in EaC2e
Support for Name Spaces

**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 runtime support for OOLs.*
Runtime Support for OOLs

Where are we?

• We have seen:
  – Symbol tables for lexically-scoped ALLs
  – Symbol tables & object tables for OOLs
  – Linkage conventions for procedure calls (in ALLs and OOLs)
  – Access links & displays to provide runtime support for lexical scopes

• ALL addressability relies on the ARP and name mangled labels
  – An OOL adds addresses that are relative to the receiving object

• Today: runtime structures to support addressability in an OOL
  – In essence, given an object fee, how do we find fee.method() and fee.member()
Compile-time Structures for OOLs — Java

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

1. Lexically scoped symbol table for the current block and its surrounding scopes in M
   - Just like an ALL — inner declarations hide outer declarations

2. Chain of symbol tables for inheritance
   - Class C and all of its superclasses
   - Need to find methods and instance variables in any superclass

3. 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 are needed for name resolution. In an ALL, we could combine 1 & 3 for a single unified set of tables. In Java, we need to split them so that the compiler can check the inheritance hierarchy between the lexical hierarchy & the global name space.
From the lecture on naming

Compile-time Structures for OOLs — Java

Conceptually

Lexical Hierarchy

Class Hierarchy

Global Scope

Search Order: lexical hierarchy, class hierarchy, global scope

Note the “sheaf of tables” model.
Class Point {
    public int x, y;
    public void draw();
}

Class ColorPoint extends Point {
    Color c;
    public void draw() {...}
    public void test() { y = x; draw(); } // override (hide) Point’s draw
}

Class C {
    int x, y;
    public void m()
    {
        Point p = new ColorPoint(); // uses ColorPoint, and, by inheritance
        y = p.x;
        p.draw(); // the definitions from Point
    }
}

This slide appeared gratuitously & pointlessly in an earlier lecture.
Runtime Structures for OOLs

Object lifetimes are independent

- Each object needs an object record (OR) to hold its state
  - Independent allocation and deallocation
    → Think “heap”
  - We will talk of OR pointers, much like AR pointers
    → We won’t call them ORPs
- Classes are objects, too
  - ORs of classes instantiate the class hierarchy

Object Records

- Distinct static storage for data members
- Need fast, consistent access
  - Known constant offsets from OR pointer
- Linguistic provision for initialization

This discussion is generic rather than Java-specific.
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

• Complex behavior arises from objects’ internal states

Ideas go back to Simula 67, & data-abstraction languages such as CLU & Alphard
Runtime Structures for OOLs

From Concept to Implementation

• Members of a class should share code members
• Replication would waste space
• Replication would be a nightmare

The Implementation

• Single copy of the code for each method
• Class has a vector of “code pointers”
• Place code vector @ fixed offset in class’ OR
Runtime Structures for OOLs

From Concept to Implementation

• Members of a class should share code members
• Replication would waste space
• Replication would be a nightmare

The Concept

The Implementation

• Single copy of the code for each method
• Class has a vector of “code pointers”
• Place code vector @ fixed offset in class’ OR
  – At offset 0 for efficiency (?)
Runtime Structures for OOLs

From Concept to Implementation

- Members of a class should share code members
- Replication would waste space
- Replication would be a nightmare

The Concept

<table>
<thead>
<tr>
<th>fee()</th>
<th>fie()</th>
<th>foe()</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
</tr>
</tbody>
</table>

The Implementation

- Single copy of the code for each method
- Move code pointer into object for faster access
  - One more slot in the OR
  - One less indirection in each call

COMP 412, Fall 2017