Chapter 6 & 7 in EaC2e
The classic example for discussing inheritance. We will use & extend it.

Class Point {
    public int x, y;
    public void draw();
}
Class ColorPoint extends Point {
    Color c;
    public void draw() {...}
    public void test() { y = x; draw(); }
}
Class C {
    int x, y;
    public void m()
    {
        Point p = new ColorPoint();
        y = p.x;
        p.draw();
    }
}

- Point is called a **base class** and ColorPoint a **derived class**
- Point is ColorPoint’s **superclass** and Colorpoint is a **subclass** of Point
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

This discussion is generic rather than Java-specific.

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
The Basics of Inheritance

**Inheritance is the primary mechanism for code reuse in OOLs**

- Class $x$ derives definitions of its members from class $y$
  - $x$ is a subclass of $y$ and $y$ is a superclass of $x$
  - Can have multi-level inheritance
- Subclass has all members of superclass
  - Includes transitive superclasses (*super-super*, ...)
- Subclass can add new members
- Subclass can override inherited definitions
  - ColorPoint inherits $x$ & $y$ from Point, adds $m$, and overrides *draw()*

**Some languages allow a class $x$ to have two or more superclasses**

- This feature is called *multiple inheritance*
- Can introduce conceptual problems with name resolution
- Languages use different rules to disambiguate these situations
Representing Inheritance at Runtime

The class hierarchy must be represented with runtime objects

Objects of class class

Plain old objects

Each class is, itself, an object

- Class, superclass & code vector pointers are data members of class class
- \texttt{LR.draw()} is found as \texttt{LR.class} \to \texttt{Point.code} + \texttt{offset} \to \texttt{LR.draw} \footnote{Code vector could exist in each object, which would make method dispatch much faster at the cost of increased space in each object. Much of the engineering on OOLs was done in small memory days.}
- Hierarchy must be built (statically, at initialization, or at class load)
Inheritance dictates OR Layout

• OR needs slots for each declared or inherited data member
  – Each object needs an instance variable that points to its class
  – Each class needs an instance variable that points to its superclass
• Can use prefixing of storage to lay out the rest of the OR
• Use the standard trick to avoid padding

Back to Our Java Example — Class Point

Class Point {
  public int x, y;
  ...
}

Class ColorPoint extends Point {
  Color c;
  ...
}

What happens if we cast a ColorPoint to a Point?
A Point to a ColorPoint?

Take the word extends literally.
Open World versus Closed World

Compile time prefixing assumes that the class structure is known when layout is performed. Two common cases occur in OO language design.

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
  - Need the compile-time data structures at runtime

C++ has a closed class structure.
Java as an open class structure.

Closed-world languages lose some dynamism.
C++ virtual functions help to recapture it.
Wait! Explain “Virtual Function”

A “virtual” function is explicitly resolved at runtime

Why does this matter?

• Assume: Point is ColorPoint’s superclass & both classes provide draw()
  – Program creates a ColorPoint \( p \)
  – Program assigns \( p \) to a Point \( q \)
  – Program invokes \( q.draw() \)
• Which \( draw() \) is invoked?

It depends on when the function name is resolved, or “bound”

• Compile-time binding would use \( q \)’s type and invoke Point’s \( draw() \)
• Runtime binding would use the \( p \)’s type & invoke ColorPoint’s draw
• In C++, the “virtual” keyword forces runtime resolution of that name
  – Virtual declaration in superclass holds for all subclasses
  – Forces a runtime search through the object’s class pointer
How does the running program find the code for a given method invocation?

Closed Class Structure

• Mapping of names to methods is static and known (C++)
  – Fixed offsets & indirect calls \( (\text{class} \rightarrow \text{code vector} \rightarrow \text{function pointer}) \)

• Virtual functions force runtime resolution

Often called “dispatch”
Runtime Resolution for Code Members

How does the running program find the code for a given method invocation?

Closed Class Structure

- Mapping of names to methods is static and known (C++)
  - Fixed offsets & indirect calls (class → code vector → function pointer)
- Virtual functions force runtime resolution

If ColorPoint inherited `draw` from Point, its code vector would refer to Point’s `draw`.

Complete code vector in each class

UL finds `draw` at offset 0 in ColorPoint’s code vector
How does the running program find the code for a given method invocation?

Closed Class Structure

• Mapping of names to methods is static
  – Fixed offsets & indirect calls (class \(\rightarrow\) class \(\rightarrow\) code vector \(\rightarrow\) function pointer)
• Virtual functions force runtime resolution

And, of course, for a little more space in each OR, we could eliminate the indirection through the class OR. (Keep a code vector in each OR)

UL finds draw at offset 0 in ColorPoint’s code vector
Runtime Resolution for Code Members

How does the running program find the code for a given method invocation?

Open Class Structure

- Dynamic mapping, unknown until runtime
- In restricted cases, can build complete code vectors at each change
  - Must lookup offsets, but can then reference into vector at a fixed offset

Superclass methods found by walking hierarchy

UL finds draw at offset 0 in ColorPoint’s code vector
Runtime Resolution for Code Members

How does the running program find the code for a given method invocation?

Open Class Structure

• Dynamic mapping, unknown until runtime
• In general case, need runtime representation of hierarchy
  — Lookup by textual name in class’ table of methods (*truly slow!*)

Superclass methods found by walking hierarchy
Runtime Resolution for Code Members

How does the running program find the code for a given method invocation?

Open Class Structure

• Dynamic mapping, unknown until runtime
• In general case, need runtime representation of hierarchy
  – Lookup by textual name in class’ table of methods (truly slow!)

If ColorPoint inherited \texttt{draw} from Point, its code vector would lack a \texttt{draw}. It would find \texttt{draw} at offset 0 in Point’s code vector.

Superclass methods found by walking hierarchy
And, in an open class structure, the code vector in each OR design creates many, many updates when a change in the class structure changes the code vector(s). (efficiency?)

**General problem:** updating ORs of instantiated objects when their class changes (language design issue)

In general case, need runtime representation of hierarchy

- Lookup by textual name in class’ table of methods (truly slow!)

Superclass methods found by walking hierarchy
Closed World versus Open World

Closed Class Structure

• Compile-time or link-time layout (ORs, classes, code vectors)
• If known at compile time, can generate static code
  – No code vector, no indirection, efficient implementation
  – Locate code the old fashioned (& fast) way
• C++ virtual functions use runtime resolution, as if open world

Open Class Structure

• Cannot lay out ORs until hierarchy is known
• With infrequent change, can change code vectors at each change
  – Single-level code vectors, fixed offsets, indirect calls
  – Instantiated objects of changed classes are still a problem
• With frequent change, may need full runtime hierarchy
  – Search for method in class hierarchy (w/tag) & cache result
  – Much more expensive, on each call or on first call
Method Calls

Given the runtime structure, how does a call work?

• Compiled code does not contain the callee’s address
• Reference is relative to receiver
  – Follow the chain: OR → class → code vector (or OR → code vector)
  – Call overhead is higher, in the best case
• Rest of pre-call, post-return, prolog, & epilog are as in an ALL

In the general case, may need dynamic dispatch

• Map code member to a search key
• Perform runtime search through hierarchy
  – This process is expensive, potentially very expensive
• Use a “method cache” to speed the search
  – Cache holds <search key, class, method pointer>

General case: Virtual functions where choice of method depends on runtime type of actual receiver — bound “late” or “at runtime.” In Smalltalk 80, all method dispatch worked this way.
Method Calls

Improvements are possible in special cases

• If class has no subclasses, can generate direct calls
  – Class structure must be static or class must be **FINAL**

• If class structure is static
  – Can generate complete method table for each class
    → Use prefixed object records and complete code vectors
  – Indirection through the class pointer or pointer in each object record
  – Keeps overhead low

• If class structure changes infrequently
  – Build complete method tables at initialization & when class structure changes

• If running program can create new classes
  – Well, not all things can be efficient
  – See Deutsch & Schiffman, POPL 1984
What is this JIT thing?

Just-in-time compilers

- JITs were developed to speed the execution of OOLs
  - Execution model is to interpret until “enough” \(^\dagger\) is known, & then compile
    - JITs capitalize on context that is discovered at runtime
  - Compiler runs during program execution
    - Compiler must, itself, be quite fast
    - Compiled code must be fast enough to cover cost of JIT execution
    - Creates an interesting set of design tradeoffs for the JIT
  - JITs have since been applied to many situations other than OOL execution

- First well known JIT was for Smalltalk 80
  - Allowed $10,000 SUN 2 to compete with $180,000 Dorado

- Most Java systems now include a JIT
  - HotSpot Server Compiler is an excellent example
  - Dynamo was a bolt-on JIT for ordinary executables

\(^\dagger\) “Enough” covers a lot of ground, from the structure of the class hierarchy & the types of objects to knowledge of which methods are frequently invoked.
What About Multiple Inheritance?

The Idea

• Let C be a subclass of both class A and class B
• C draws some, but not all, methods from class A
• C draws some, but not all, methods from class B

Need a linguistic mechanism to specify partial inheritance

Problems arise when C inherits from both A & B

• C’s OR can extend A or B, but not both
• C’s code vector can extend A or B, but not both
• Some class D might inherit from either A or B or both
  – Need consistency in OR layout & code vector layout
• Both A & B might provide fum() — which is seen in C?
  – C++ produces a “syntax error” when fum() is used

Java 8 adds a limited form of multiple inheritance with default methods.

Need a better way to say “inherit”
We can try to use prefixing again

Class Point {
    public int x, y;
    public void draw();
    public Point add();
}

Class CThing {
    Color c;
    public void colorize();
}

Class CPoint extends Point and CThing {
    public void draw();
}

How do casts work?

- If we cast a CPoint as a Point, x & y are at the desired offsets
- If we cast a CPoint as a CThing, c is at the wrong offset
  - Extra class field is the key
  - Cast bumps “self” to point at 2\textsuperscript{nd} class field

Class that occurs out of “single inheritance prefix order” needs additional information — class field.
Assume 64 bit slots in the ORs

OR Layout with Multiple Inheritance

Need to prefix both OR & code vector

Class Point {
    public int x, y;
    public void draw();
    public Point add();
}

Class CThing {
    Color c;
    public void colorize();
}

Class CPoint extends Point and CThing {
    public void draw();
}

COMP 412, Fall 2017

But, where does CPoint’s colorize find “c”?
OR Layout with Multiple Inheritance

Solution: Prefix both code storage & code vector

Class Point {
    public int x, y;
    public void draw();
    public Point add();
}

Class CThing {
    Color c;
    public void colorize();
}

Class CPoint extends Point and CThing {
    public void draw();
}

“c” is at offset 8 in a CThing
“c” is at offset 32 in a CPoint
To fix CPoint’s offset for “c”, use a trampoline function

Class Point {
    public int x, y;
    public void draw();
    public Point add();
}

Class CThing {
    Color c;
    public void colorize();
}

Class CPoint extends Point and CThing {
    public void draw();
}