Object Oriented Languages
Object-Oriented Languages

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

- Each object needs local storage for its fields
  - Fields are static (lifetime of object)
  - Often access through methods

- Methods are interfaces outside objects
  - Some methods are public, others are private
  - More complex linkage convention; locating them is complex
Class vs. Object

- Objects with the same data layout & code members
  - Same names for instance variables & methods
  - But, separate instance variables for each class instance
  - Share codes and static data (a.k.a. class variables)

- Class is a type, object is an instance of the class type

- Benefits of OOL design
  - Allows programmer to write it down once
  - Allows code reuse at both source & implementation level
Accessibility and Inheritance

- What data an executing method can see?
  - *instance variables*
    - The object’s own per-instance variables
  - *class variables*
    - The class’s own per-class variables (static fields)
  - Any object defined in the global name space (or scope)
    - Objects may contain other objects (or references to them)

- Inheritance
  - Most OOLs support a hierarchical notion of inheritance
  - Some OOLs support multiple inheritance
    - More than one path through the inheritance hierarchy
Issues in Implementing OOLs

- Two critical issues in OOL implementation:
  - Object representation
  - Mapping a method invocation name to a method implementation
  - Both are intimately related to the OOL’s name space

- Object Representation
  - Private storage for instance variables
    - Objects (or instances) allocated in heap
    - Need consistent, fast access: constant offsets
  - Static class storage for class variables accessible by global names
    - Accessible via linkage symbol &C.count (e.g. class C::count)
    - Nested classes are handled like blocks in Algol-Like-Languages
  - Method code put at fixed offset from start of class area
    - Maintain pointers to method codes
Dealing with Single Inheritance

- Use **prefixing** of storage

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

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

Does casting work properly?
Resolving Method Names

- Mapping names to methods
  - `<class, method> ⇒ method implementation`
  - Static mapping, known at compile-time (Java, C++)
    - Fixed offsets & indirect calls
  - Dynamic mapping, unknown until run-time (Smalltalk)
    - Look up name in class’ table of methods
    - Dynamic class hierarchy

- This is really a data-structure problem
  - Build a table of function pointers (method table for each class)
  - Use a standard invocation sequence
    - Read function address from the fixed entry of method table
    - Invoke indirect call
Per-Class Method Table

- With static, compile-time mapped classes

Message dispatch becomes an indirect call through a method table
Dispatching in Single Inheritance

- Use **prefixing** of tables: fixed entry for same name

```
Class Point {
    int x, y;
    public void draw();
    public void d2o();
}
```

```
Class ColorPoint extends Point {
    Color c;
    public void draw();
    public void rev();
}
```
To simplify object creation,

- We allow a class to inherit methods from `superclass`.
- The descendant class is called the `subclass` of its ancestor.

The Concept:

Method tables of `B` & `C` are extensions of the table from `A`
Static vs. Dynamic Inheritance

- Two distinct philosophies

<table>
<thead>
<tr>
<th>Static class hierarchy</th>
<th>Dynamic class hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can map <a href="">class:method</a> to code at compile time</td>
<td>Cannot map <a href="">class:method</a> to code at compile time</td>
</tr>
<tr>
<td>Leads to 1-level jump vector</td>
<td>Multiple jump vector (one per class)</td>
</tr>
<tr>
<td>Copy superclass methods</td>
<td>Must search method tables</td>
</tr>
<tr>
<td>Fixed offsets &amp; indirect calls</td>
<td>Run-time lookups &amp; caching</td>
</tr>
<tr>
<td>Less flexible &amp; expressive</td>
<td>Much more expensive to run</td>
</tr>
</tbody>
</table>

- Visibility in name space
  - Method can see instance variables of self class & superclasses
  - Many different levels where a value can reside

- In essence, OOL differs from ALL in
  - the shape of its name space AND
  - the mechanism used to bind names to implementations
Multiple Inheritance

- **The idea**
  - Allow more flexible sharing of methods & attributes
  - Relax the inclusion requirement
  - Need a linguistic mechanism for specifying partial inheritance

- **Problems when C inherits from both A & B**
  - C’s method table can extend A or B, but not both
    - Layout of an object instance for C becomes tricky
  - Other classes, say D, can inherit from C & B
    - Adjustments to offsets become complex
  - Both A & B might provide fum() with the same name
    - which is seen in C?
    - C++ produces a “syntax error” when fum() is used
Multiple Inheritance - fields

- Use prefixing of storage

```java
Class Point {
    int x, y;
}

Class ColoredThing {
    Color c;
}

Class ColorPoint extends Point, ColoredThing {
}
```

Does casting work properly?
Multiple Inheritance - fields & methods

- Use prefixing of storage

Class Point {
    int x, y;
    void draw();
    void d2o();
}

Class CThing {
    Color c;
    void rev();
}

Class CPoint extends Point, CThing {
    void draw()
}

Use as CThing
Multiple Inheritance (casting & method call)

- **Usage as Point:**
  - No extra action (prefixing does everything)

- **Usage as CThing:**
  - Increment `self` by 12

- **Usage as CPoint:**
  - Lay out Cthing’s class pointer and Cthing’s data at `self + 12`
  - When calling `rev()`
    - All methods has a pointer `self` as an implicit parameter
    - Two possible options
      - Add 12 to `self` in pre-call and restore `self` in post-call sequences
      - The call in class table points to a *trampoline function* that adds 12 to `self` and calls `rev()`
    - Ensures that `rev()`, which assumes that `self` points to a CThing data area, gets the right data
Multiple Inheritance (trampoline function)

- Assume that C inherits fee() from A, fie() from B, & defines both foe() and fum()
- Object record for an instance of C
  - Method table entry for fie() contains a point to a trampoline function instead of real pointer to B::fie()
  - Trampoline function increases self pointer and call B::fie()
What About Calls in an OOL (Dispatch)?

- In an OOL, most calls are indirect calls
  - Compiled code does not contain address of callee
    - Finds it by indirection through class’ method table
    - Required to make subclass calls find right methods
    - Code compiled in class C cannot know of subclass methods that override methods in C and C’s superclasses

- In a general case, need dynamic dispatch
  - Map method name to a search key
  - Perform a run-time search through hierarchy
    - Start with object’s class, search for 1st occurrence of key
    - This can be expensive
  - Use a method cache to speed up the search
    - Cache holds <key, class, method pointer>
What About Calls in an OOL (Dispatch)?

- Improvements are possible in special cases
  - If class has no subclasses, can generate direct call
    - Class structure must be static or class must be `FINAL`
  - If class hierarchy is static
    - Can generate complete method table for each class
    - Single indirection through class pointer (**1 or 2 operations**)
    - Keeps overhead at a low level
  - If class hierarchy changes infrequently
    - Build complete method tables at run time
    - Initialization & any time class hierarchy changes
  - If running program can create new classes, ...
    - Well, not all things can be done quickly
Summary

- OOLs support inheritance
  - Single vs. multiple inheritance
  - Casting to superclasses

- Issues in implementing OOLs
  - Data layout
  - Method mapping

- Optimization for method invocation
  - Direct call based on class hierarchy analysis