Multiple Inheritance

Basically, MI means that a class may extend or inherit directly from more than one parent class.

→ esp. useful when modelling “real-world” situations

![Class Hierarchy Diagram]

Sounds like a good idea, eh?

BUT ...

→ there are a variety of technical problems with MI.

→ MI is often grossly misused in practise, and it’s acquired something of a bad reputation.

→ many (older) OO languages support only single inheritance.

→ C++ and most recent OO languages support MI.

→ Java doesn’t support MI per se

→ however, you can get a lot of the power of MI using Java interfaces

So What’s the Problem?

- Most problems with MI stem from trying to find the “appropriate” implementation of a method, but finding more than one.

- *i.e.*, most of the problems stem from allowing multiple inheritance of *method implementations*

  → “Which version of draw() will be used here?”

- With single inheritance only (*e.g.*, Java), inheritance defines a tree and you just look up the parent chain until you find a method definition. No confusion!

  → If you allow MI (*e.g.*, C++), which branch do you look up?

- Allowing multiple inheritance of abstract methods only (*i.e.*, Java interfaces) is much less confusing (to both human reader and the compiler).

  → Only one method implementation can be inherited.

MI: Name Clashes

→ Which draw does GraphicalCowboy inherit? Can you have both somehow?

Solutions:

1. Use precedence list to disambiguate.

2. Insist on overriding when ambiguities arise.
MI: Inheritance from a Common Ancestor

A special case of the name clash problem:

- Do we really want two copies of the all of features from ListElement?

**Ans:** Possibly, but probably not.

---

MI: Creeping Featuritis

- Bad programming habit commonly observed in C++ world:
  - Want to adapt several ideas into one monster new class.
  - Therefore just inherit from several different classes you are interested in.
- Better idea:
  - Be careful how you design your classes.
  - Break into manageable, distinct pieces rather than throwing everything in one big basket.

---

What Java Does

- Each class can extend (inherit from) one parent.
  [If you don’t specify, then you inherit from java.lang.Object by default]
  This is the only class you can inherit method implementations from. Therefore there is never any confusion about which method implementation to use.
- Java also supports the idea of interfaces: All variables must be final (i.e., constants) and all methods must be abstract.
A java class can implement any number of interfaces! It must provide implementations for all methods it acquires (or else be declared as as abstract class).

Main advantage is increased polymorphism:

→ You can treat an object of class D as if it were an A, a B, or a C.
→ You can pass it as an argument to any method expecting an A, a B, or a C as a parameter.

Main advantage is increased polymorphism:

→ You can treat an object of class D as if it were an A, a B, or a C.
→ You can pass it as an argument to any method expecting an A, a B, or a C as a parameter.

An Example Using Interfaces

Let us suppose we have a really great method to sort elements of an array.

We sort numbers, based on numeric values and the mathematical operator <, but suppose we would like to be able to sort arbitrary kinds of objects, as long as they supply as idea of lessThan.

→ For strings, use alphabetic ordering.
→ For Figures, use area.
→ For Employee records, use SSN.

Sounds like a job for inheritance!

[Well, interfaces actually.]

Define an interface called Sortable.

One abstract method, lessThan.

abstract boolean lessThan (Sortable s);

Define Sort method to sort an array of Sortables.

public static void sort (Sortable[] A);

Can pass in an array of any class that implement Sortable!

Any class you want to be able sort an array of should implement the interface and provide a definition for lessThan.

Can now use CoolSort.sort!

Here’s the code for the Sortable and CoolSort:

```java
public interface Sortable {
    abstract boolean lessThan (Sortable s);
}

public class CoolSort {
    // This implements a technique called ShellSort.
    public static void sort (Sortable[] A) {
        int n = A.length, incr = n/2, i;
        while (incr >= 1) {
            for (i=incr; i<n; i++) {
                Sortable temp = A[i];
                int j = i;
                while (j>= incr
                        && temp.lessThan(A[j-incr])) {
                    A[j] = A[j-incr];
                    j = j - incr;
                }
                A[j] = temp;
            }
            incr = incr / 2;
        }
    }
}
```
class Employee implements Sortable {
    // Social security number.
    private int SSN;

    // returns true iff my SSN is less than
    // the other Employee's SSN.
    public boolean lessThan (Sortable s) {
        Employee otherEmployee = (Employee) s;
        return this.SSN < otherEmployee.SSN;
    }
    ...
}

class EmployeeDB {
    private Employee[] database;

    public void addEmployee (Employee e) {...
    public void deleteEmployee (Employee e) {...

    public void sortDB () {
        CoolSort.sort(database);
    }
    ...
}

abstract class Figure implements Sortable {
    // returns true iff my area is less than
    // the other Figure's area.
    public boolean lessThan (Sortable s) {
        Figure otherFigure = (Figure) s;
        return this.area() < otherFigure.area();
    }
    ...
}

class TestFigures {
    public static void main (String [] args) {
        Figure[] figureList = new Figure[4];
        figureList[0] = new Circle();
        figureList[0].setSize (10);
        figureList[1] = new Rectangle();
        figureList[1].setSize (100, 30);
        figureList[2] = new Square();
        figureList[2].setSize (10);
        figureList[3] = new Square();
        figureList[3].setSize (5);
        // continued on next slide...
    }
    ...
}

// continued from previous slide...
if (figureList[0].lessThan (figureList[1])) {
    System.out.println("The first figure is " + "smaller than the second.");
} else {
    System.out.println("The first figure is " + "NOT smaller than the second.");
}
int i;
System.out.println("Figure list before sorting:");
for (i=0; i<figureList.length; i++) {
    figureList[i].draw();
    System.out.println(" + area = " + figureList[i].area());
}
CoolSort.sort (figureList);
System.out.println("Figure list after sorting:");
for (i=0; i<figureList.length; i++) {
    figureList[i].draw();
    System.out.println(" + area = " + figureList[i].area());
}
}

Just a Minute Here ... 

Why couldn’t we make Sortable an abstract class and just have Figure and Employee extend Sortable?

Well we could, but:

→ The kind of thing we want to sort might already inherit from another class. And Java doesn’t permit multiple inheritance.

→ Most interfaces express an abstract property that may be shared across many otherwise-unrelated classes (which may also inherit from another class already).

→ Usually, an interface represents only one aspect or possible use of a class.

→ Many Java interface names, therefore, end in “able”:

→ Cloneable, Serializable, Runnable

[Also 1.1 Java AWT (i.e., the Java GUI toolkit) adds “Listener” interfaces to handle various kinds of GUI things.]
• This approach allows us to create generic routines for sorting, etc. which we keep in a library for general use.

→ If you want to be able to CoolSort, then make sure your classes implement the Sortable interface and define lessThan.

→ If you do that, then you can use CoolSort right from the library.

• Think of an interface as being:

  - generic object + some interesting property

→ Sortable represents a generic object that can be sorted
[Not every class has an obvious idea of what lessThan should mean.]

---

The Java Enumeration Interface

java.util Enumeration is an interface that may be implemented by any class.

```java
public interface Enumeration {
    public abstract boolean hasMoreElements();
    public abstract Object nextElement();
}
```

java.util.Enumeration defines a method called elements that returns an enumeration of the vector.

```java
public class Vector {
    public final synchronized Enumeration elements();
    ...
}
```

---

Notes on Enumerations

• OK, you can actually walk through a Vector easily enough using the elementAt method:

```java
for (int i=0; i<tricks.size(); i++) {
    Card c = tricks.elementAt(i);
    numPointsThisRound += c.value();
}
```

... BUT the idea of an enumeration is more general:

→ If you have a complicated data structure (e.g., a tree), you can define an abstract way of iterating through the elements so that others can, say, check or print the value of each node.

→ Your implementation of nextElement can decide in what order the nodes will be visited. The user doesn’t have to worry or know about the hairy details in order to “walk through” the data structure.
The Iterator Design Pattern

Intent:

→ Provide a clean, abstract way to access all of the elements in an aggregate (container) without exposing the underlying representation.

→ Also, moves responsibility for access and traversal to a separate “iterator” object.

Motivation:

→ Often want to say to a container (e.g., tree, graph, table, list, graphic):
  "Apply f() to each of your objects."

→ Don’t want to expose implementation details of the container AND want to allow multiple simultaneous traversals

⇒ Create separate interface/class that provides simple “hooks”.

Notes on the Iterator Design Pattern

• The iterator interface basically provides two hooks:
  – “Next, please.”
  – “All done?”

Can provide a few others hooks too (how many left, go back, start over, etc.)

• Can define different iterator implementation that redefine what “next element” means.

⇒ walk through a tree, alphabetical order, reverse alphabetical order, order of insertion, LRU

Can have multiple iterations ongoing simultaneously (enumerator state basically consists how far along it is in the “list”).

• Of course, if you change (or allow others to change) the aggregate (add/remove) or its contained objects, then chaos may result! Therefore:

⇒ May want to lock out changes during an iteration (concurrency control).

⇒ Iterator may take fast snapshot of aggregate state and use that to iterate through.

⇒ May want to buffer requested changes while an iterator is active, then commit.

• Complicated structures may be difficult to iterate through.

⇒ Store “path taken” or “nodes visited”.

OO Design Patterns

Ref: Design Patterns, Elements of Reusable Object-Oriented Software
by Gamma, Helm, Johnson, & Vlissides
[aka The Gang of Four]
Addison-Welsley, 1995.

Basic idea: Reusable metaphors for designing OO software systems.

I highly recommend you buy and read this book, but wait a year or two until you have more experience in programming. It will mean more to you then.
Object-Level vs. Class-Level Visibility

- **Object-level visibility** — [OOTuring] An object cannot examine/alter private parts of other instances of the same class.

- **Class-level visibility** — [Java, C++] An object is implicitly a “friend” to all other objects of the same class.

  i.e., Can access the private parts of other objects of the same class passed as parameters.

An Example

```java
public class Complex {
    private double re, im;
    public Complex (double re, double im) {
        this.re = re;
        this.im = im;
    }
    public Complex add (Complex c) {
        // Can access c.re and c.im!
        Complex ans = new Complex
            ((this.re + c.re, this.im + c.im);
        return ans;
    }
    public String toString () {
        return re + " " + im + "i";
    }
}

public static void main (String [] args) {
    Complex c1 = new Complex(17, 4);
    Complex c2 = new Complex(-3.5, 4);
    System.out.println ("c1 + c 2=+ c1.add(c2));
}
```

Inheritance and IS-A

One the one hand, we often view inheritance as modelling the IS-A ("is a kind of") relationship:

```
mammal
    
    primate
    
    human
    
    rodent
    
    chimp
    
    rat
    
    mouse
```

Continuity of “shape”:

- all variables/functions of parent inherited in child
- BUT you expect some differentiation too
  [else why distinguish?]
  → parent’s abstract functions implemented in child
  → sometimes add new “features”
  → idea is that child specializes parent but any instance of child IS-A instance of parent too.
Inheritance as Subtyping

- Principle of substitutability: derived classes from a common base class should be conceptually interchangeable with each other
  
  → Semantic continuity: [Godfrey]
    Method $foo()$ should behave in roughly the same way across all of the descendants.
  
  → Assumed/stated properties of the parent should be preserved in the children.

- Counter examples:
  
  e.g., $\text{Circle.setSize(int, int)}$
  
  e.g., $\text{Circle.draw()}$ renders a gif of Dogbert

- Usually, parent methods are either fully or partially defined (e.g., implicit pre-conditions); child specializes some of these, perhaps by wrapping a few lines of code around a called to the parent’s method.

When to Subclass

specialization — child is a specialized version of the parent class. Parent’s methods are usually fully-defined.
  
  e.g., chimp inherits from primate
  
  - Most common (and best) reason to subclass.
  
  - Uses inheritance to model the abstract IS-A relationship.

extension — child extends behaviour of parent, adding new attributes/methods.
  
  e.g., ExtendedStack inherits from Stack
  
  - Basically, an implementation technique to achieve re-use.
  
  - Common when using third-party libraries; often need to inherit and customize.

Inheritance as Subtyping

- Preserves/models IS-A relationship

- Most common reason to subclass

- Some think that this (plus inheritance-as-specification) is the one true religion: inheritance is subtyping and vice versa.
  
  → C++ FAQ author calls it “good inheritance”, everything else “bad inheritance”

They’re wrong, although well intentioned.

generalization — child overrides its parent to create a more general class.

  e.g., a threeDCircle class that extends circle to allow both solid and wireframe circles to be drawn.

[Extension adds new functionality; generalization overrides existing functionality.]

- Many “purists” feel this is usually wrong; that you’ve got an inverted hierarchy that needs redesigning.

- However, often such a re-organization is either impractical or impossible (esp. if it’s someone else’s code).
specification — parent class defines syntax and structure of all descendant classes, but some methods are abstract.

e.g., a general Stack class that has children ArrayStack and ListStack that implement the abstract methods push and pop.

- Abstract base classes allow you to define the “shape” of all derived classes.
  ⇒ key to large-scale abstraction

combination — several parent classes are inherited from to create a new child class with features of all parents.

[requires multiple inheritance and/or interfaces]

- Sometimes, this is the correct thing to do.
- Often, however, this is done carelessly as a quick hack to get desired extra functionality with minimal work. In this case, it would be better to carefully consider whether HAS-A is the appropriate relationship.

OO Summary

- OO programming has proven to be a great success in industry.
  — Support for information hiding, inheritance, re-use of existing code.
  — It’s getting easier to cobble systems together from other people’s components.
- ... however, it’s really a tool for “scaling up” a level of complexity. You can always write bad OO code.
  — In fact, object-oriented languages make it easier to write even bigger, more incomprehensible systems!
  — Still working on engineering principles for how to do it “right”.

CS211 — OOSUMMARY