

CASE STUDY

OBJECT-ORIENTED PROGRAMMING

OUTLINE

- Prelude: Abstract Data Types
- The Object Model (Ada)
- Smalltalk

Ask not what you can do
for your classes,
Ask what your classes can do
for you.

Owen Astrachan
Duke University



PRELUDE: ABSTRACT DATA TYPES

- Imperative programming paradigm
 - Algorithms + Data Structures = Programs [Wirth]
 - Produce a program by functional decomposition
 - Start with function to be computed
 - Systematically decompose function into more primitive functions
 - Stop when all functions map to program statements

PROCEDURAL ABSTRACTION

- Concerned mainly with interface
 - Function
 - What it computes
 - Ignore details of how
 - Example: sort(list, length);

DATA ABSTRACTION

- Or: abstract data types
- Extend procedural abstraction to include data
 - Example: type float
- Extend imperative notion of type by:
 - Providing encapsulation of data/functions
 - Example: stack of int's
 - Separation of interface from implementation

ENCAPSULATION

- **Definition:** *Encapsulation* is a mechanism which allows logically related constants, types, variables, methods, and so on, to be grouped into a new entity.
- Examples:
 - Procedures
 - Packages
 - Classes

SIMPLE STACK IN C

```
#include <stdio.h>

struct Node {
    int val;
    struct Node* next;
};

typedef struct Node* STACK;

STACK stack = NULL;

int empty( ) {
    return stack == NULL;
}

int pop( ) {
    STACK tmp;
    int rslt = 0;
    if (!empty()) {
        rslt = stack->val;
        tmp = stack;
        stack = stack->next;
        free(tmp);
    }
    return rslt;
}

void push(int newval) {
    STACK tmp = (STACK)malloc(sizeof(struct Node));
    tmp->val = newval;
    tmp->next = stack;
    stack = tmp;
}

int top( ) {
    if (!empty())
        return stack->val;
    return 0;
}
```

A STACK TYPE IN C

```
struct Node {  
    int val;  
    struct Node* next;  
};  
typedef struct Node* STACK;  
  
int empty(STACK stack);  
STACK newstack( );  
int pop(STACK stack);  
void push(STACK stack, int newval);  
int top(STACK stack);
```

GOAL OF DATA ABSTRACTION

- Package
 - Data type
 - Functions
- Into a module so that functions provide:
 - public interface
 - defines type

GENERIC PROGRAMMING IN ADA

generic

```
    type element is private;
```

```
package stack_pck is
```

```
    type stack is private;
```

```
    procedure push (in out s : stack; i : element);
```

```
    procedure pop (in out s : stack) return element;
```

```
    procedure isempty(in s : stack) return boolean;
```

```
    procedure top(in s : stack) return element;
```

- Similar to C++ templates

```
private
  type node;
  type stack is access node;
  type node is record
    val : element;
    next : stack;
  end record;
end stack_pck;
```

```
package body stack_pck is
procedure push (in out s : stack; i : element) is
temp : stack;
begin
temp := new node;
temp.all := (val => i, next => s);
s := temp;
end push;
```

```
procedure pop (in out s : stack) return element is
    temp : stack;
    elem : element;
begin
    elem := s.all.val;
    temp := s;
    s := temp.all.next;
    dispose(temp);
    return elem;
end pop;
```

```
procedure isempty(in s : stack) return boolean is
begin
    return s = null;
end isempty;
```

```
procedure top(in s : stack) return element is
begin
    return s.all.val;
end top;
end stack_pck;
```

THE OBJECT MODEL

- Problems remained:
 - Automatic initialization and finalization
 - No simple way to extend a data abstraction
- Concept of a class
- Object decomposition, rather than function decomposition

CLASS

- **Definition:** A *class* is a type declaration which encapsulates constants, variables, and functions for manipulating these variables.
- A class is a mechanism for defining an ADT.

```
class MyStack {  
    class Node {  
        Object val;  
        Node next;  
        Node(Object v, Node n) { val = v;  
            next = n; }  
    }  
    Node theStack;  
  
    MyStack( ) { theStack = null; }  
  
    boolean empty( ) { return theStack == null; }
```

```
Object pop( ) {  
    Object result = theStack.val;  
    theStack = theStack.next;  
    return result;  
}  
}
```

```
Object top( ) { return theStack.val; }
```

```
void push(Object v) {  
    theStack = new Node(v, theStack);  
}  
}
```

CONCEPTS IN OOP

- Constructor
- Destructor
- Client of a class
- Class methods (Java static methods)
- Instance methods

CONCEPTS IN OOP (II)

- OO program: collection of objects which communicate by sending messages
 - A invokes a method of B and pass params
 - A waits for return values from B
- Generally, only 1 object is executing at a time
- Object-based language (vs. OO language)
 - no support of subtyping (inheritance)
- Classes
 - Determine type of an object
 - Permit full type checking

VISIBILITY

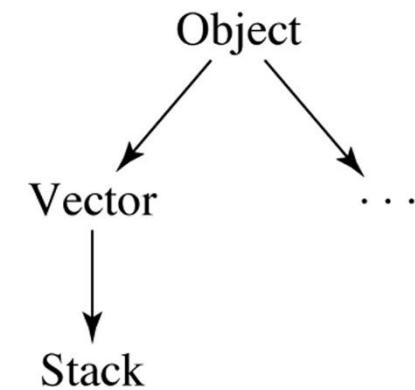
- public
- protected
- private

INHERITANCE (SUBTYPING)

- Class hierarchy
 - Subclass, parent or super class
- is-a relationship
 - A stack *is-a* kind of a list
 - So are: queue, deque, priority queue
- has-a relationship
 - Identifies a class as **a client of another class**
 - Aggregation
 - A class is an aggregation if it contains other class objects

INHERITANCE (II)

- In single inheritance, the class hierarchy forms a tree.
- Rooted in a most general class: *Object*
- Inheritance supports code reuse
- Remark: in Java a *Stack* extends a *Vector*
 - Good or bad idea?
 - Why?
- Single inheritance languages: Smalltalk, Java



MULTIPLE INHERITANCE

- Allows a class to be a subclass of zero, one, or more classes.
- Class hierarchy is a directed graph
- Advantage: facilitates code reuse
- Disadvantage: more complicated semantics
 - Re: *Design Patterns* book mentions multiple inheritance in conjunction with only two of its many patterns.

OBJECT ORIENTED LANGUAGE

- **Definition:** A language is *object-oriented* if it supports
 - an encapsulation mechanism with information hiding for defining abstract data types,
 - virtual methods, and
 - inheritance

POLYMORPHISM

- Polymorphic - having many forms
- **Definition:** In OO languages *polymorphism* refers to the late binding of a call to one of several different implementations of a method in an inheritance hierarchy.

- Consider the call: obj.m();
 - obj of type T
 - All subtypes must implement method m()
 - In a statically typed language, verified at compile time
 - Actual method called can vary at run time depending on actual type of obj
- Subtyping polymorphism

```
for (Drawable obj : myList)
    obj.paint( );
// paint method invoked varies
// each graphical object paints itself
// essence of OOP
```

POLYMORPHISM (CONT'D)

- **Definition:** A subclass method is *substitutable* for a parent class method if the subclass's method performs the same general function.
- Thus, the *paint* method of each graphical object must be transparent to the caller. E.g.,
 - Button
 - Panel
 - Choice Box
- The code to paint each graphical object depends on the principle of *substitutability*.

TEMPLATES OR GENERICS

- A kind of class generator
- Can restrict a Collections class to holding a particular kind of object
- **Definition:** A *template* defines a family of classes parameterized by one or more types.
- Prior to Java 1.5, clients had to downcast an object retrieved from a Collection class.
- **Parametric polymorphism:** $\forall A. A \rightarrow A$

```
ArrayList<Drawable> list = new ArrayList<Drawable> ();
```

```
...
```

```
for (Drawable d : list)  
    d.paint(g);
```

ABSTRACT CLASSES

- **Definition:** An *abstract class* is one that is either declared to be abstract or has one or more abstract methods.
- **Definition:** An *abstract method* is a method that contains no code beyond its signature.

- Any subclass of an abstract class that does not provide an implementation of an inherited abstract method is itself abstract.
- Because abstract classes have methods that cannot be executed, client programs cannot initialize an object that is a member of an abstract class.
- This restriction ensures that a call will not be made to an abstract (unimplemented) method.

EXPRESSION ABSTRACT SYNTAX

```
abstract class Expression { ... }
```

```
    class Variable extends Expression { ... }
```

```
abstract class Value extends Expression { ... }
```

```
    class IntValue extends Value { ... }
```

```
    class BoolValue extends Value { ... }
```

```
    class FloatValue extends Value { ... }
```

```
    class CharValue extends Value { ... }
```

```
class Binary extends Expression { ... }
```

```
class Unary extends Expression { ... }
```

INTERFACES

- **Definition:** An *interface* encapsulates a collection of constants and abstract method signatures.
- An interface may not include either variables, constructors, or non-abstract methods.
- Difference between interface and abstract classes:
 - Interface:
 - All methods must be abstract
 - Only constants
 - Abstract class:
 - Some methods can be implemented
 - Objects can be declared

```
public interface Map {  
    public abstract boolean containsKey(Object key);  
    public abstract boolean containsValue(Object value);  
    public abstract boolean equals(Object o);  
    public abstract Object get(Object key);  
    public abstract Object remove(Object key);  
    ...  
}
```

INTERFACE AND MULTIPLE INHERITANCE

- Because it is not a class, an interface does not have a constructor, but an abstract class does.
- Some like to think of an interface as an alternative to multiple inheritance.
- Strictly speaking, however, an interface is not quite the same since it doesn't provide a means of reusing code; i.e., all of its methods must be abstract.
- An interface is similar to multiple inheritance in the sense that an interface is a type.
- A class that implements multiple interfaces appears to be many different types, one for each interface.

VIRTUAL METHOD TABLE (VMT)

- How the appropriate virtual method is called at run time.
- At compile time the actual run time class of any object may be unknown.

```
MyList myList;
```

```
...
```

```
System.out.println(myList.toString());
```

VMT (CONT'D)

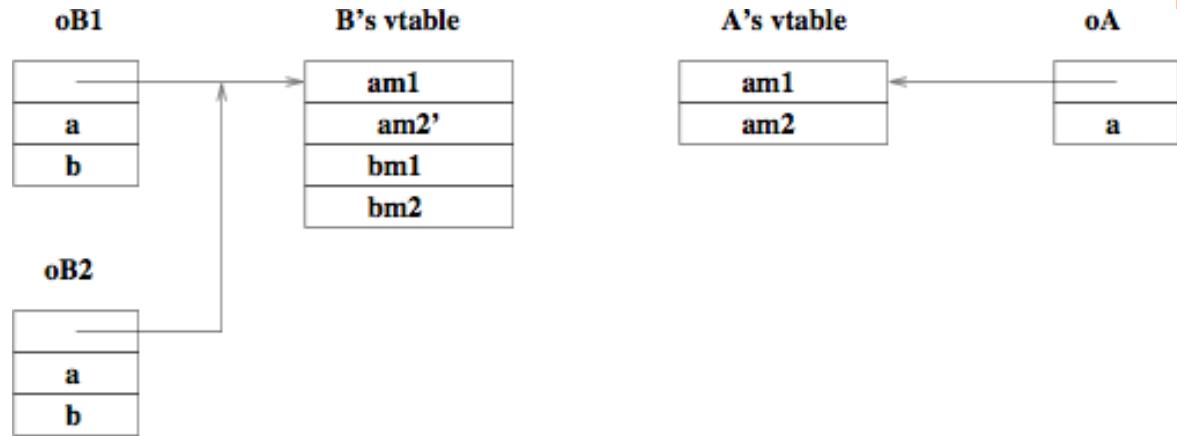
- Each class has its own VMT, with each instance of the class having a reference (or pointer) to the VMT.
- A simple implementation of the VMT would be a hash table, using the method name (or signature, in the case of overloading) as the key and the run time address of the method invoked as the value.
- For statically typed languages, the VMT is kept as an array.
- The method being invoked is converted to an index into the VMT at compile time.

```

class A {
    Obj a;
    void am1( ) { ... }
    void am2( ) { ... }
}

class B extends A {
    Obj b;
    void bm1( ) { ... }
    void bm2( ) { ... }
    void am2( ) { ... }
}

```



RUN TIME TYPE IDENTIFICATION

- **Definition:** Run time type identification (RTTI) is the ability of the language to identify at run time the actual type or class of an object.
- All dynamically typed languages have this ability, whereas most statically typed imperative languages, such as C, lack this ability.
- At the machine level, recall that data is basically untyped.

- In Java, for example, given any object reference, we can determine its class via:
- `Class c = obj.getClass();`

REFLECTION

- Reflection is a mechanism whereby a program can discover and use the methods of any of its objects and classes.
- Reflection is essential for programming tools that allow plugins (such as Eclipse -- www.eclipse.org) and for JavaBeans components.

- In Java the *Class* class provides the following information about an object:
 - The superclass or parent class.
 - The names and types of all fields.
 - The names and signatures of all methods.
 - The signatures of all constructors.
 - The interfaces that the class implements.

```
Class class = obj.getClass( );
Constructor[ ] cons = class.getDeclaredConstructors( );
for (int i=0; i < cons.length; i++) {
    System.out.print(class.getName( ) + "(" );
    Class[ ] param = cons[i].getParameterTypes( );
    for (int j=0; j < param.length; j++) {
        if (j > 0) System.out.print(", ");
        System.out.print(param[j].getName( );
    }
    System.out.println( ")" );
}
```

SMALLTALK

- The original object-oriented language
- Developed in 1970s at Xerox PARC
- Xerox Alto
 - Smalltalk system
 - OS
 - IDE
 - mouse based GUI
 - Steve Jobs visit → Macintosh

GENERAL CHARACTERISTICS

- Simple language
- Most of the class libraries written in Smalltalk
- Everything is an object, even control structures
- Excluding lexical productions, grammar has 21 production rules (3 pages)

- The value of every variable is an object; every object is an instance of some class.
- A method is triggered by sending a message to an object.
 - The object responds by evaluating the method of the same name, if it has one.
 - Otherwise the message is sent to the parent object.
 - The process continues until the method is found; otherwise an error is raised.
- All methods return a value (object).

- Precedence
 - Unary messages, as in: `x` negated
 - Binary messages, as in: `x + y`
 - Keyword messages, as in: `Turtle go: length`
- In the absence of parentheses, code is evaluated from left to right.

- Examples:
 - $x + y * z$ squared
 - a max: b - c
 - anArray at: i put: (anArray at: i + 1)
- By default, Smalltalk uses infinite precision, fractional arithmetic.
 - $1/3 + 2/6 + 3/9$ evaluates to 1.

```
(a > b) ifTrue: [ max := a ]  
ifFalse: [ max := b ].
```

- [] - uninterpreted block
- A block is like an object, too
- Boolean methods: **ifFalse:** and **ifTrue:**
- **ifTrue:** If the object is the true object, it executes the code block it has been handed. If it is the false object, it returns without executing the code block.
- **ifFalse:** symmetrical

BLOCKS

sum := 0.

1 to: n do: [:i | sum := sum + (a at: i)].

sum := 0.

a do: [:x | sum := sum + x].

sum := 0.

i := 1.

[i <= n] whileTrue: [

 sum := sum + (a at: i).

 i := i + 1].

"True methods"

ifTrue: trueBlock ifFalse: falseBlock

 ^ trueBlock value

ifTrue: aBlock

 ^ aBlock value

ifFalse: aBlock

 ^ nil

ifFalse: falseBlock ifTrue: trueBlock

 ^ trueBlock value

^ means return

EXAMLE: POLYNOMIALS

- Represent Polynomials: $3x^2 + 5x - 7$
- Representation: #(-7 5 3)
- Subclass of Magnitude

Magnitude subclass: #Polynomial

instanceVariableNames: 'coefficient'

classVariableNames:: "

poolDictionaries: "

new

"Unary class constructor: return 0*x^0"

^ self new: #(0)

new: array

"Keyword class constructor"

^ (super new) init: array

init: array

"Private: initialize coefficient"

coefficient := array deepCopy

degree

"Highest non-zero power"

^ coefficient size - 1

coefficient: power

"Coefficient of given power"

(power >= coefficient size) ifTrue: [^ 0].

^ coefficient at: power + 1

asArray

 ^ coefficient deepCopy

= aPoly

 ^ coefficient = aPoly asArray

!= aPoly

 ^ (self = aPoly) not

< aPoly

 "not defined"

 ^ self shouldNotImplement