Concepts of Programming Languages

Lecture 10 - Object-Oriented Programming

Patrick Donnelly

Montana State University

Spring 2014

Administrivia

Reading:

Chapter 12

I am surprised that ancient and modern writers have not attributed greater importance to the laws of inheritance . . .

Alexis de Tocqueville (1840)

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

Owen Astrachan

Object-oriented Languages

Many object-oriented programming (OOP) languages

Some support procedural and data-oriented programming (e.g., Ada 95+ and C++)

Some support functional program (e.g., CLOS)

Newer languages do not support other paradigms but use their imperative structures (e.g., Java and C#)

Some are pure OOP language (e.g., Smalltalk & Ruby)

Some functional languages support OOP, but they are not discussed in this chapter

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

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

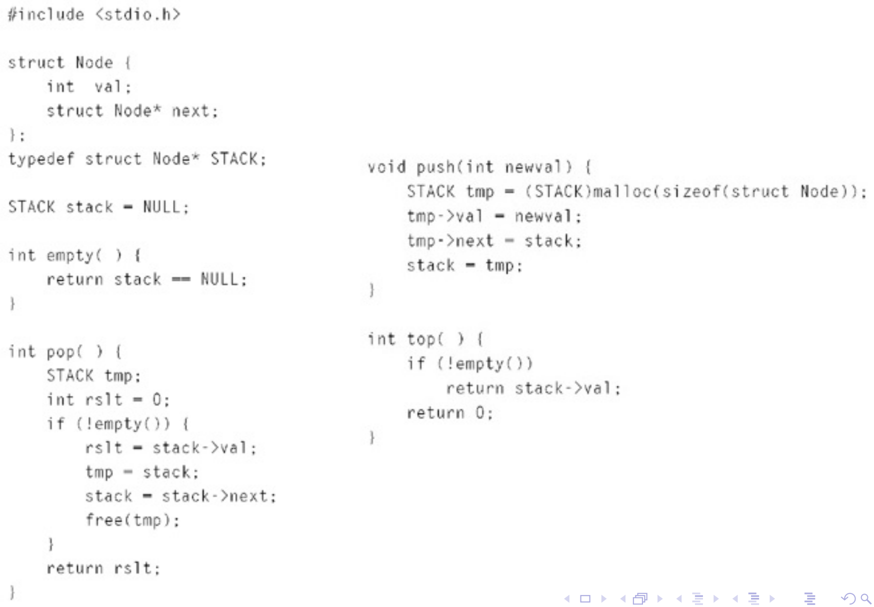
Definition

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

A 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);
```

Implementation of Stack Type in C

```
#include "stack.h"
#include <stdio.h>

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

STACK newstack( ) {
    return (STACK) NULL;
}

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

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

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

Goal of Data Abstraction

Package

- Data type
- Functions

Into a module so that functions provide:

- public interface
- defines type

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 abstract data type.

Simple Stack Class in Java (1/2)

```
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; }
```

Simple Stack Class in Java (2/2)

```
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);
    }
}
```

Object Model

Constructor

Destructor

Client of a class

Class methods (Java static methods)

Instance methods

Object Model

OO program: collection of objects which communicate by sending messages

Generally, only 1 object is executing at a time

Object-based language (vs. OO language)

Classes

- Determine type of an object
- Permit full type checking

Visibility

public

protected

private

Inheritance

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

In single inheritance, the class hierarchy forms a tree.

Rooted in a most general class: Object

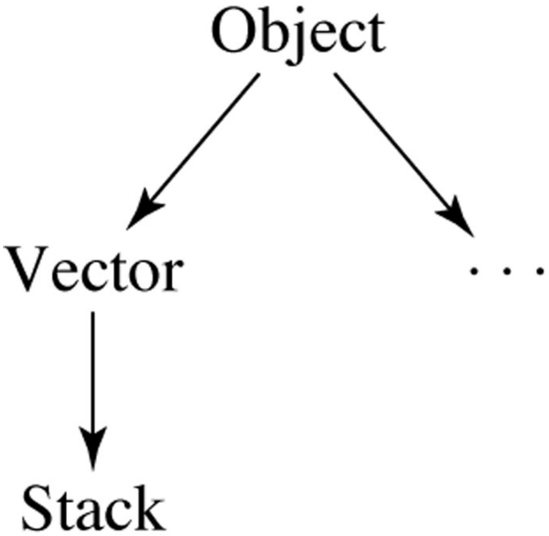
Inheritance supports code reuse

Remark: in Java a Stack extends a Vector

- Good or bad idea?
- Why?

Single inheritance languages: Smalltalk, Java

Inheritance


$$\begin{tikzcd} & \text{Object} \arrow[dl] \arrow[dr] & \\ \text{Vector} \arrow[d] & & \cdots \\ \text{Stack} & & \end{tikzcd}$$

Inheritance

Calls to methods are called *messages*

The entire collection of methods of an object is called its *message protocol* or *message interface*

Messages have two parts–a method name and the destination object

In the simplest case, a class inherits all of the entities of its parent

Inheritance

Inheritance can be complicated by access controls to encapsulated entities

- A class can hide entities from its subclasses
- A class can hide entities from its clients
- A class can also hide entities for its clients while allowing its subclasses to see them

Besides inheriting methods as is, a class can modify an inherited method

- The new one overrides the inherited one
- The method in the parent is overriden

Inheritance

Three ways a class can differ from its parent:

1. The parent class can define some of its variables or methods to have private access, which means they will not be visible in the subclass
2. The subclass can add variables and/or methods to those inherited from the parent
3. The subclass can modify the behavior of one or more of its inherited methods.

Inheritance

There are two kinds of variables in a class:

- Class variables - one/class
- Instance variables - one/object

There are two kinds of methods in a class:

- Class methods - accept messages to the class
- Instance methods - accept messages to objects

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

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.

Polymorphism

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`

Polymorphism Example

```
for (Drawable obj : myList)
        obj.paint( );

// paint method invoked varies
// each graphical object paints itself
// essence of OOP
```

Dynamic Binding

Definition

A ***polymorphic variable*** can be defined in a class that is able to reference (or point to) objects of the class and objects of any of its descendants

When a class hierarchy includes classes that override methods and such methods are called through a polymorphic variable, the binding to the correct method will be dynamic

Allows software systems to be more easily extended during both development and maintenance

Substitutability

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.

The code to paint each graphical object depends on the principle of substitutability.

Polymorphism

Essence: same call evokes a different method depending on class of object

Example: `obj.paint(g);`

- Button
- Panel
- Choice Box

Substitutability principle

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.

Generics Example

```
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.

Abstract Classes

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 an abstract class.

This restriction ensures that a call will not be made to an abstract (unimplemented) method.

Abstract Classes Example

```
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 { ... }
```

Nested Classes

If a new class is needed by only one class, there is no reason to define so it can be seen by other classes

- Can the new class be nested inside the class that uses it?
- In some cases, the new class is nested inside a subprogram rather than directly in another class

Other issues:

- Which facilities of the nesting class should be visible to the nested class and vice versa

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.

Interface Example

```
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);
    ...
}
```

Interfaces

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.

Interfaces

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.

The Exclusivity of Objects

Everything is an object

- Advantage - elegance and purity
- Disadvantage - slow operations on simple objects

Add objects to a complete typing system

- Advantage - fast operations on simple objects
- Disadvantage - results in a confusing type system (two kinds of entities)

Include an imperative-style typing system for primitives but make everything else objects

- Advantage - fast operations on simple objects and a relatively small typing system
- Disadvantage - still some confusion because of the two type systems

Virtual Method Table (VMT)

How is 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( ));
```

Virtual Method Table (VMT)

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.

Virtual Method Table (VMT)

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.

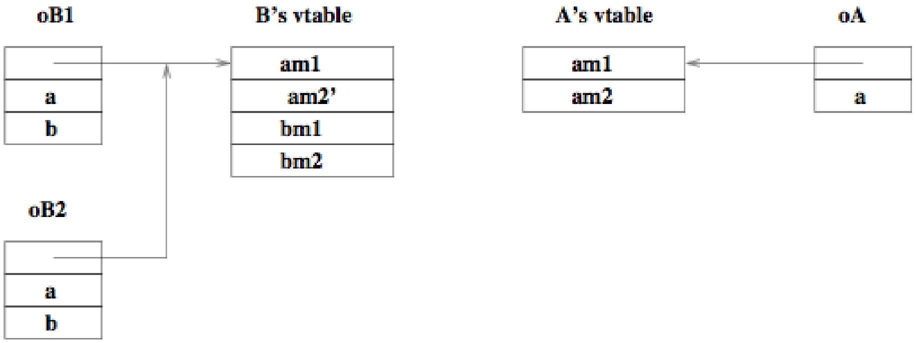
The method being invoked is converted to an index into the VMT at compile time.

Virtual Method Table (VMT) Example

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

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

Virtual Method Table (VMT) Example



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

Definition

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.

Reflection

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.

Reflection Example

```
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( ")" );
}
```

Summary

Smalltalk is a pure OOL

C++ has two distinct type systems (hybrid)

Java is not a hybrid language like C++; it supports only OOP

C# is based on C++ and Java

Ruby is a relatively recent pure OOP language; provides some new ideas in support for OOP

Summary

OO programming involves three fundamental concepts: ADTs, inheritance, dynamic binding

Major design issues: exclusivity of objects, subclasses and subtypes, type checking and polymorphism, single and multiple inheritance, dynamic binding, explicit and implicit de-allocation of objects, and nested classes

Implementing OOP involves some new data structures