Chapter 13
Object Oriented Programming

Contents

13.1 Prelude: Abstract Data Types

Imperative programming paradigm

- *Algorithms + Data Structures = Programs*
- *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*

Defn: *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

Figure 13.1

```
#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 Simple Stack in C

```
#include <stdio.h>
struct Node {
    int val;
    struct Node* next;
};
typdedef struct Node* STACK;
STACK theStack = NULL;
```

```
int empty( ) { return theStack == NULL; }


int pop ( ) {
    STACK temp = theStack;
    int result = theStack->val;
    theStack = theStack->next;
    free(temp);
    return result;
}
```

```
int top( ) { return theStack ->val; }


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

Problems

Users cannot declare a variable of type stack.

- *The code defines only a single instance of a stack.*

Restricted to a stack of integers.

A Stack Type in C with Better Encapsulation

Figure 13.2

stack.h

```
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

Figure 13.3

stack.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;
}
```

Now, users can declare a variable of type stack.

just need to include the header file.

Problems

Permit accesses as a linked list rather than a real stack.

Restricted to a stack of integers.

Not easy to extend the type.

Goal of Data Abstraction

Package

- *Data type*
- *Functions*

Into a module so that functions provide:

- *public interface*
- *defines type*

Ada introduces packages.

```
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;
```

```
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;
```

13.2 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

Defn: 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 (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);
    }
}
```

- OO program: collection of objects which communicate by sending messages

Defn: A language is *object-oriented* if it supports

- an encapsulation mechanism with information hiding for defining abstract data types,
- virtual methods, and
- inheritance

Some Terms

- Classes
  - *Determine type of an object*
  - *Permit full type checking*
- Method: function in OO.
- Class constructor: allocate heap space and initialize the object.
- Class destructor: final operations before destroying the object.

Some Terms (cont.)

Client of a class C

- *Any other class or method that declares or uses an object of class C*

Class methods (Java static methods)

- *invoked through the class name*
  - exception: constructor is invoked through the *new* operation.

Instance methods

- *invoked through an object*

Inner class

- *A class defined inside another class*

```
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);
    }
}
```

Visibility of a variable or method

- public
  - *visible to any client and subclass of the class*
- protected
  - *visible only to a subclass of the class*
  - *(Java has a special meaning of protected classes:*
    - visible to every class in the same package)
- private
  - *visible only to the current class*

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*

Single inheritance

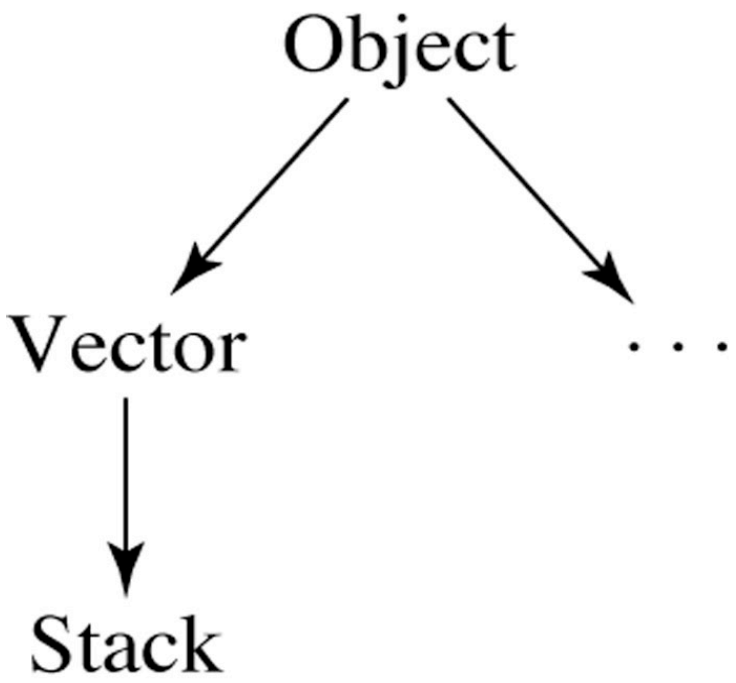
- *A subclass can have only one parent class*
- *The class hierarchy hence forms a tree.*

Rooted in a most general class: *Object*

Inheritance supports code reuse

Single inheritance languages: Smalltalk, Java

Inheritance of Stack in Java:



Good or bad design?
Why?

Alternative design:
Stack has a private vector object.

```
Class Stack {
    private Vector stack;
    public push(...){...}
    public pop(...) {...}
}
```

A common practice

- *Make instance variables private*
- *Allow accesses only through the use of public or protected methods*

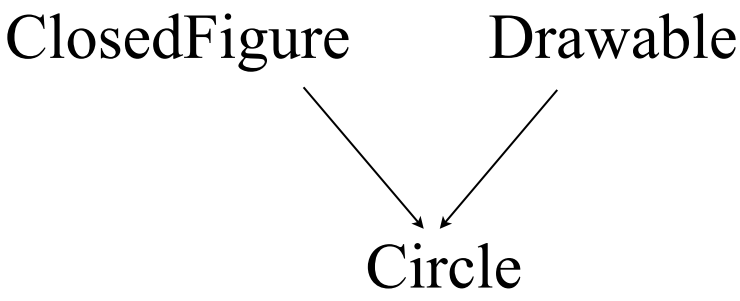
Good or bad? Why?

Minimize needed changes if the class is modified.

Multiple Inheritance

- Allows a class to be a subclass of zero, one, or more classes.
- Class hierarchy is a directed graph
- Examples: C++, Python

Example



- Adv: facilitates code reuse
- Disadv: more complicated semantics
  - A method is defined in both parent classes, which one is inherited by the child?
    - Python uses left-to-right, depth-first search
      - search child, then parent1 and its super classes, then parent2 and its super classes, and so on.

Polymorphism

Polymorphic - having many forms

Defn: In OO languages, it's late binding of a call to one of several different implementations of a method in an inheritance hierarchy.

Virtual Method:

A method whose behavior can be overridden within an inheriting class by a function with the same signature.

In Java, by default, all methods are virtual.

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

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

Defn: 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.

Templates or Generics

A kind of class generator

Can restrict a Collections class to holding a particular kind of object

Defn: A *template* defines a family of classes parameterized by one or more types.

```
ArrayList<Drawable> list = new ArrayList<Drawable> ();
...
for (Drawable d : list)
    d.paint(g);
```