Chapter 13
Object-Oriented Programming

I am surprised that ancient and Modern writers have not attributed greater importance to the laws of inheritance ...  
Alexis de Tocqueville

Contents

13.1 Prelude: Abstract Data Types
13.2 The Object Model
13.3 Smalltalk
13.4 Java
13.5 Python
Ask not what you can do
for your classes,
Ask what your classes can do
for you.

Owen Astrachan

Procedural Abstraction

_Procedural abstraction_ allows the programmer to be concerned mainly with the interface between the function and what it computes, ignoring the details of how the computation is accomplished.

- _Function_
- _What it computes_
- _Ignore details of how_
- Example: sort(list, length);
13.1 Prelude: Abstract Data Types

Good examples of data abstraction occur among the primitive data types.

- *Float*: encapsulates standard arithmetic operations (addition, subtraction, etc.)
- *Abstract* because the use of type *float* does not depend on the concrete representation of its values or the details of how the operations are computed.
- Based on the idea of *encapsulation*.
- Extends the idea of *procedural abstraction* to include *data abstraction*.

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

```c
#include <stdio.h>

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

STACK stack = NULL;

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

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

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*

*Figure 13.1*
A Stack Type in C

Figure 13.2

```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

Figure 13.3

```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 ret = 0;
    if (empty()) {
        ret = stack->val;
        tmp = stack;
        stack = stack->next;
        free(tmp);
    }
    return ret;
}

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

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

The key goal of data abstraction is to:

- Package the data type and its functions together into a single module so that the functions provide a public interface to the type.
- Modules/packages allow the programmer to restrict access to:
  - The underlying concrete representation of the abstract data type.
  - The implementation details of its functions.

```plaintext
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

Solved by introducing the 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 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);
    }
}

• Constructor
• Destructor
• Client of a class C – any other class or method that declares or uses an object of class C.
• Class methods (Java static methods)
• Instance methods

• 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

*Information hiding* is made explicit by requiring that every method and instance variable in a class have a particular level of visibility.

*Public* – visible to any client and subclass of the class.

*Protected* – visible to only a subclass of the class.

*Private* – visible to the current class, but not to its subclasses or clients

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 NOT as a subclass*
  - *Aggregation*
  - *A class is an aggregation if it contains other class objects*
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
- Adv: facilitates code reuse
- Disadv: more complicated semantics
- Re: Design Patterns book mentions multiple inheritance in conjunction with only two of its many patterns.
- Multiple inheritance language: C++, Python

**Defn:** 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

**Defn:** 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` – this is referred to as *dynamic binding* of method calls
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.
Polymorphism

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

Ex: 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

**Defn:** 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.
ArrayList<Drawable> list = new ArrayList<Drawable> ();
...
for (Iterator i = myList.iterator(); i.hasNext(); ) {
    Drawable obj = (Drawable) i.next();
    obj.paint();
}

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

**Defn:** An *abstract class* is one that is either declared to be abstract or has one or more abstract methods.

**Defn:** 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 an abstract class.

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

**Defn:** An *interface* encapsulates a collection of constants and abstract method signatures.  
An interface may not include either variables, constructors, or non-abstract methods.
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);
    ...
}

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 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( ));
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

Defn: 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:

```java
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](http://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( ")" );
}

13.4 Java

• 1.0 released in 1995
• 1.5 (or Java 5) in 2004
• major language changes: 1.1, 1.5
• steady growth in SE libraries
• mixed language
  – *primitive types:* int, double, boolean
  – *objects*
• statically typed
• single inheritance
Direct support for:

- inner classes
- visibility modifiers
- abstract classes
- interfaces
- generics
- run time type identification
- reflection