Object-Oriented Programming
Contents

13.1  Prelude: Abstract Data Types
13.2  The Object Model
13.3  Smalltalk
13.4  Java
13.5  Python
13.1 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*
**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

```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

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 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*
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 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
- 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

• public
• protected
• private

Public methods and instance variables in a class constitute by default the class’ interface to external 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

• not to be confused with 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

In *single* inheritance, the class hierarchy forms a tree.

Rooted in a most general class: *Object*

Inheritance supports code reuse

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
• + facilitates code reuse
• - more complicated semantics
• Re: *Design Patterns* book mentions multiple inheritance in conjunction with only two of its many patterns.
Object-oriented vs. Object-based

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

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

We’ll call it subtype polymorphism to distinguish it from the parametric polymorphism of functional languages.
Consider the call: obj.m();

- obj of type T
- All subtypes must implement method m()
- In a statically typed language, this is verified at compile time
- If m() can be re-implemented by a subclass it is called a virtual method
- 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
Substitutability Principle

**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.
Subtype Polymorphism

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

Ex: obj.paint(g);
   - Button
   - Panel
   - Choice Box

Substitutability principle
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.
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

**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.
Interfaces as types

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.
Templates or Generics

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

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

Prior to Java 1.5, clients had to downcast an object retrieved from a Collection class.

Generics are Java’s version of parametric polymorphism
ArrayList<Drawable> list = new ArrayList<Drawable> ();

... 

for (Drawable d : list)
    d.paint(g);
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 am2() { ... }  // reimplements am2
    void bm1() { ... }
    void bm2() { ... }
}
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( ");
}