CIS 343 - Structure of Programming Languages

Nathan Bowman

Slides by (with slight modifications): Ira Woodring

Object-Oriented Programming (Follows the Sebesta Text Chapter 12)

Abstract Data Types (ADTs)

Although ADTs are just one facet of OOP, it is difficult to find a modern language that implements ADT but is not object-oriented

This is not a bad thing, but rather a testament to the success of OOP

Abstract Data Types (ADTs)

ADT + inheritance + dynamic method binding = OOP

Abstract Data Types (ADTs)

Inheritance is a powerful idea that allows reuse of code in related ADTs (*classes* in OOP parlance)

Because we have experience in OOP, it can be hard to separate the ideas of ADTs and inheritance, but try to imagine what it would really be like to have ADTs but no inheritance

Abstract Data Types (ADTs)

With inheritance:

- Write useful base class
- Create subclass that is mostly the same, just with a few methods overridden
- Code can be heavily reused

Without inheritance:

- Write useful class
- Write a different class for similar task that duplicates almost all of the code

Abstract Data Types (ADTs)

Most modern OO languages allow only single inheritance, meaning that a class has just one parent class

C++ allows a subclass to inherit from multiple parents, or even from no parents. This is more flexible, but is generally considered more trouble than it is worth

Abstract Data Types (ADTs)

A compromise, seen in e.g., Java and C#, is to use **interfaces**

An interface specifies a set of methods that must be implemented by implementing classes, but does not include any implementations itself

Classes can implement multiple interfaces without the complexity that comes from multiple inheritance

Abstract Data Types (ADTs)

What about **dynamic method binding**?

Recall that "dynamic" means "at runtime"

The code for a particular method call is not necessarily bound (decided on) until runtime

Abstract Data Types (ADTs)

This is useful in relation to inheritance

```
class Animal:
    speak():
        ...
    ...
```

```
class Dog(Animal):
    speak():
        ...
    ...
```

```
Animal a = Dog();
a.speak();
```

Abstract Data Types (ADTs)

In static method binding, when you call function `foo()`, the compiler knows exactly what function you really want

In dynamic method binding, when you call `some_animal.speak()`, the system needs to wait until the call happens to see what kind of animal is really speaking

Abstract Data Types (ADTs)

One implementation detail to be aware of when using OOP is how/where objects are allocated

Can be any of

- static
- dynamic on the heap
- dynamic on the stack

Some languages allow all three

Abstract Data Types (ADTs)

Also need to be aware of how *deallocation* works

Some languages (such as C++) require the programmer to deallocate memory manually

It is more popular in modern languages to have a **garbage collector** that deallocates resources automatically, though this does have some drawbacks

Abstract Data Types (ADTs)

Let's view some examples!

We'll start with Ada:

Abstract Data Types (ADTs)

Ada uses **packages** for encapsulation. Packages have two parts:

- **package specification:** which provides the interface
- **body package:** which provides the implementation

Both of these parts are also called packages.

Packages are generalized encapsulations; they can define multiple types.

Abstract Data Types (ADTs)

Both the specification and the body package share the same name, but the reserved keyword body in the package header notes it is the body package.

The two parts can be compiled separately (as long as the specification is compiled first).

Abstract Data Types (ADTs)

The programmer of a type can decide if the representation is hidden or not.

If it is not hidden, is this an ADT?

NO!

If we can manipulate the underlying data, it is not an ADT.

Abstract Data Types (ADTs)

In ADA there are two ways of hiding information. The first is to use the keyword `private`, the other is to define the ADT as a pointer in the package specification and that pointer points to a structure defined in the body package.

Since the implementation or body of the package is hidden from clients, the clients can't see how it is implemented.

Ada Package Specification

```
package Stack_Pack is
-- The visible entities, or public interface
  type Stack_Type is limited private;
  Max_Size : constant := 100;
  function Empty(Stk : in Stack_Type) return Boolean;
  procedure Push(Stk : in out Stack_Type;
    Element : in Integer);
  procedure Pop(Stk : in out Stack_Type);
  function Top(Stk : in Stack_Type) return Integer;

  -- The part that is hidden from clients
  private
  type List_Type is array (1..Max_Size) of Integer;
  type Stack_Type is
    record
    List : List_Type;
```

Ada Body specification

```
with Ada.Text_IO; use Ada.Text_IO;
package body Stack_Pack is

function Empty(Stk: in Stack_Type) return Boolean is begin
  return Stk.Topsub = 0;
  end Empty;

procedure Push(Stk : in out Stack_Type; Element : in Integer) is
  begin
  if Stk.Topsub >= Max_Size then
      Put_Line("ERROR - Stack overflow");
    else
      Stk.Topsub := Stk.Topsub + 1;
      Stk.List(Topsub) := Element;
    end if;
end Push;
```

Abstract Data Types (ADTs)

You may have noticed the keywords `with` and `use` in the packages.

- `with` makes names defined in external packages visible (includes or imports them).
- `use` makes it so we don't need to qualify the symbols explicitly (just like the `using` keyword in C++).

Abstract Data Types (ADTs)

Here is client code that makes use of the Stack.

```
with Stack_Pack;
use Stack_Pack;
procedure Use_Stacks is
  Topone : Integer;
  Stack : Stack_Type;   -- Creates an Object of type Stack_Type
  begin
  Push(Stack, 42);
  Push(Stack, 17);
  Topone := Top(Stack);
  Pop(Stack);
  ...
end Use_Stacks;
```

Abstract Data Types (ADTs)

C++ also allows ADTs, through use of the class construct. This differs from Ada; because Ada uses the more generalized encapsulation technique, if a program makes use of a package it has access to all of the public symbols defined in that package.

C++ use of the class construct means that when we are a client to an ADT we only have access to the ADT information.

Abstract Data Types (ADTs)

C++ classes define data called **members** and functions called **member functions**. These members are of the type **instance** or **class**.

Class members are associated with the class (think `static`).

Instance members share class member functions but have their own member data.

Abstract Data Types (ADTs)

Member functions in C++ can be defined in the class implementation file or just the header file.

If a function is defined (not just declared) in the header it is implicitly inlined.

An inline function is a function that the compiler replaces the function call with the function body. In some situations this can save time (consider if the function is small, it might be quicker than creating a new stack frame).

Abstract Data Types (ADTs)

C++ classes have `private` and `public` sections to provide information hiding (also `protected` which is discussed later).

C++ Interface file

```
#ifndef IntCell_H
#define IntCell_H

/**
 * A class for simulating an integer memory cell.
 */
class IntCell
{
  public:
    explicit IntCell( int initialValue = 0 );
    int read( ) const;
    void write( int x );

  private:
    int storedValue;
};
```

C++ Implementation file (notice the namespace operator):

```
#include "IntCell.h"

/**
 * Construct the IntCell with initialValue
 */
IntCell::IntCell( int initialValue ) : storedValue{ initialValue
{
}

/**
 * Return the stored value.
 */
int IntCell::read( ) const
{
    return storedValue;
}
```

C++ All in header example:

```
#include <iostream>

/**
 * A class for simulating an integer memory cell.
 */
class IntCell
{
  public:
    /**
     * Construct the IntCell.
     * Initial value is 0.
     */
    IntCell( )
      { storedValue = 0; }

    /**
     * Construct the IntCell.
     * Initial value is initialValue.
     */
    IntCell( int initialValue )
      { storedValue = initialValue; }

    /**
     * Return the stored value.
     */
    int read( )
      { return storedValue; }

    /**
     * Change the stored value to x.
     */
    void write( int x )
      { storedValue = x; }
```

```
  private:
    int storedValue;
};
```

Abstract Data Types (ADTs)

We would then use the class as such:

```
Intcell iCell;
iCell.write(42);
```

Abstract Data Types (ADTs)

Java supports ADTs using the class construct. In Java all objects are declared on the heap and accessed (implicitly) through references. As noted in lecture, the syntactic unit for a Java class is a single file.

```
class StackClass {
        private int [] stackRef; private int maxLen,
                topIndex;
        public StackClass() { // A constructor
                stackRef = new int [100]; maxLen = 99;
                topIndex = -1;
        }
        public void push(int number) {
                if (topIndex == maxLen)
                        System.out.println("Error in push—stack i
                else stackRef[++topIndex] = number;
        }

        public void pop() {
                if (empty())
                        System.out.println("Error in pop—stack is
```

Client:

```
public class TstStack {
  public static void main(String[] args) {
    StackClass myStack = new StackClass(); myStack.push(42);
    myStack.push(29);
    System.out.println("29 is: " + myStack.top()); myStack.pop();
    System.out.println("42 is: " + myStack.top());
    myStack.pop();
    myStack.pop();  // Produces an error message
  }
}
```

Abstract Data Types (ADTs)

C# is similar to both Java and C++, but adds new features.

Like Java, C# keeps objects on the heap, and provides both classes and structs. The difference in both is that the default access modifier for a class is private, whereas for a struct it is public. Additionally, structs do not support inheritance, and are value types instead of reference types (put on the stack!).

Abstract Data Types (ADTs)

C# provides properties that allow getters and setters to be generated.

C# also has no requirement that a class must reside in a single file; you can create a `partial` class.

As with C++, you may provide more than one class definition per file (though this is not good practice).

```
class TimePeriod
{
    private double seconds;

    public double Hours
    {
        get { return seconds / 3600; }
        set { seconds = value * 3600; }
    }
}


class Program
{
    static void Main()
    {
        TimePeriod t = new TimePeriod();

        // Assigning the Hours property causes the 'set' accessor to be called.
        t.Hours = 24;

        // Evaluating the Hours property causes the 'get' accessor to be called.
        System.Console.WriteLine("Time in hours: " + t.Hours);
    }
}
// Output: Time in hours: 24
```

Abstract Data Types (ADTs)

Ruby also provides ADTs, defined between a `class` keyword and the `end` keyword.

Instance variables are prefaced with the @ sign, and instance methods begin with `def` and end with `end`.

Class variables are denoted with @@. There are multiple ways to create a class method, such as

```ruby
class Blah
  def self.myMethod
    ...
  end
end

class Blah
  class << self
    def myMethod
      ...
    end
  end
end

class Blah; end
def Blah.myMethod
```

Abstract Data Types (ADTs)

Constructors are named `initialize` in Ruby and cannot be overloaded.

Class members may be added to the class at any time, and class methods may be removed.

These abilities greatly harm the readability of the language.

Abstract Data Types (ADTs)

Access controls are also dynamic, meaning that violations cannot be caught before runtime.

The default access is public for methods, but Ruby provides for protected and private as well.

For instance data access is private, and this cannot be changed. We access instance data through the use of accessor methods called attributes.