Abstract Data Types

Abstraction

- Abstraction
  - View of an entity where only most significant attributes are included
  - Generalization, stereotype
- Mechanism to deal with complexity
- Process abstraction
  - Process can be used without knowing implementation
  - Subroutines
- Data abstraction
  - Object can be used without knowing its representation or the implementation of its operations
  - E.g. classes

Data Abstraction

- Early history
  - COBOL records
  - C structs
  - SIMULA 67 allowed association of representation with implementation
- Abstract data type
  - Set of data values (objects)
    - Typically specify representation
  - Set of operations
    - Subroutines
    - Typically enclosed in an encapsulation unit
    - Information hiding
- Primitive types as ADTs
User-defined ADTs

- **ADT**
  - Representation of objects hidden
  - Only defined operations available
  - Representation and operations in a single syntactic unit
  - Interface does not depend on representation
  - Clients cannot declare variables of the ADT type
- **Information hiding**
  - Client cannot manipulate representation intentionally or by accident
  - Representation variables can only be accessed/modified over small regions of code
  - Reduces name conflicts and size of namespace
  - Substitutability of different representation/implementations
  - Separate compilation

Design Issues

- **Syntactic unit** for encapsulation
  - Defines a type
  - Encloses representation and implementation
  - Provides access to operations
- **Built-in operations**
  - Few
- **Access controls**
  - Getters/setters
- **Separation of interface and implementation**
- **Constructors/destructors**
- **Parametric ADTs**

Example – Stack ADT

- **create(stack)**: Creates and possibly initializes a stack object
- **destroy(stack)**: Deallocates the storage for the stack
- **empty(stack)**: A predicate for Boolean function that returns true if the specified stack is empty and false otherwise
- **push(element, stack)**: Pushes the specified element on the specified stack
- **pop(stack)**: Removes the top element from the specified stack
- **top(stack)**: Returns a copy of the top element from the specified stack
Ada

- package
  - Specification
  - Body
  - May contain type declarations
  - Separate compilation
- Information hiding
  - public/private parts in specification
  - Representation/implementation in body and type is pointer
- Built-in operations
  - private types have assignment and equality
  - limited private types have no built-in operations
- Evaluation
  - One of first languages to support ADTs
  - package can define a type but is not one

```ada
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; return Integer);
  -- The part that is hidden from client
  private
  type List_Type is array (1..Max_Size) of Integer;
  type Stack_Type is record
    List : List_Type;
    TopIndex : Integer range 0..Max_Size := 0;
  end record;
end Stack_Pack;
```

```ada
with Ada.Text_IO; use Ada.Text_IO;
package body Stack_Pack is
  -- Function: Empty returns Boolean value
  function Empty(STK : in Stack_Type) return Boolean is
    if STK.TopIndex > 0 then
      return False;
    else
      return True;
    end if;
  end Empty;

  -- Procedure: Push adds an item to the stack
  procedure Push(STK : in out Stack_Type; Element : in Integer) is
    if STK.TopIndex < STK.Max_Size then
      STK.List(STK.TopIndex) := Element;
      STK.TopIndex := STK.TopIndex + 1;
    else
      return Error(3721);
    end if;
  end Push;

  -- Procedure: Pop removes an item from the stack
  procedure Pop(STK : in out Stack_Type) is
    if STK.TopIndex > 0 then
      STK.List(STK.TopIndex) := STK.List(STK.TopIndex) - 1;
      STK.TopIndex := STK.TopIndex - 1;
    else
      return Error(3722);
    end if;
  end Pop;

  -- Function: Top returns the top element of the stack
  function Top(STK : in Stack_Type) return Integer is
    return STK.List(STK.TopIndex);
  end Top;

end Stack_Pack;
```
C++

- Added object-orientation to C
- **class**
  - Encapsulation unit and defines a type
  - Instances can be static, stack-dynamic or heap-dynamic
  - Specification can be in a header
    - Separate compilation
    - Inlining of methods in class definition
- Information hiding
  - **public/private** members
- Constructors/destructors
  - Called implicitly
- Evaluation
  - Supports ADTs, information hiding and separate compilation
```cpp
void main() {
    int topnum;
    Stack stk;  // Create an instance of the Stack class
    stk.push(17);
    topnum = stk.top();
    stk.pop();
    ...}

class Stack {
private:   //** These members are visible only to other
//** members and friends (see Section 11.6.4)
    int stackNum;
    int maxlen;
    int topnum;
    friends:  //** These members are visible to clients
    stack(); //** A constructor
    ~Stack(); //** A destructor
    void push(int);
    void pop();
    int top();
    int empty();
}

// Stack.h - the header file for the Stack class
#include <iostream.h>

class Stack {
private:   //** These members are visible only to other
//** members and friends (see Section 11.6.4)
    int stackNum;
    int maxlen;
    int topnum;
    friends:  //** These members are visible to clients
    stack(); //** A constructor
    ~Stack(); //** A destructor
    void push(int);
    void pop();
    int top();
    int empty();
};

// Stack.cpp - the implementation file for the Stack class
#include <iostream.h>
namespace Stack {
    Stack(int)
    { //** A constructor
        stackNum = maxlen = 0;
        topnum = 0;
    }
    Stack() [delete] stack(); //** A destructor
    void push(int)
    { //** A constructor
        if (stackNum < maxlen)
            stackNum = stackNum + 1;
        else
            throw Stack::SizeException();
        topnum = stackNum;
    }
    void pop()
    { //** A destructor
        if (topnum > 0)
            topnum = topnum - 1;
        else
            throw Stack::EmptyException();
    }
    int top() [throw Stack::EmptyException]
    { //** A destructor
        return topnum;
    }
    int empty() [throw Stack::EmptyException]
    { //** A destructor
        return stackNum == 0;
    }
};
```
Java, C#

- **Java**
  - Similar to C++
  - All objects are heap-dynamic
  - Interface can define without implementation
  - *class* must have complete implementation
  - Access modifiers on each member
  - No destructors

- **C#**
  - Similar to Java
  - Default constructors/destructors
  - Getters/setters

```java
public class StackClass {
    private int[] stack;
    private int top;

    public StackClass() { // constructor
        stack = new int[10];
        top = -1;
    }

    public void push(int value) {
        if (top < stack.length - 1) { // if stack is not full
            stack[++top] = value;
        } else {
            System.out.println("Error: stack is full");
        }
    }

    public int pop() {
        if (top == -1) { // if stack is empty
            System.out.println("Error: stack is empty");
            return -1;
        } else {
            return stack[top--];
        }
    }

    public int size() {
        return top + 1;
    }

    public void displayStack() {
        for (int i = 0; i <= top; i++) {
            System.out.print(stack[i] + "\n");
        }
    }
}
```

```csharp
public class StackClass {
    private int[] stack;
    private int top;

    public StackClass() { // constructor
        stack = new int[10];
        top = -1;
    }

    public void push(int value) {
        if (top < stack.length - 1) { // if stack is not full
            stack[++top] = value;
        } else {
            Console.WriteLine("Error: stack is full");
        }
    }

    public int pop() {
        if (top == -1) { // if stack is empty
            Console.WriteLine("Error: stack is empty");
            return -1;
        } else {
            return stack[top--];
        }
    }

    public int size() {
        return top + 1;
    }

    public void displayStack() {
        for (int i = 0; i <= top; i++) {
            Console.WriteLine(stack[i] + "\n");
        }
    }
}
```
Objective-C

- Added object-orientation to C using Smalltalk syntax
- Interface/implementation
- Method notation
  - Declaration
  - Parameters
  - Call (Message passing)
- Initializers
  - Explicit call
- Information hiding
  - @public/@private directives
  - Methods are public
  - Getters/setters
  - Properties
- Evaluation
  - Explicit call to constructors
  - All methods public
  - Mixed syntax

```
@interface class-name : parent-class
  
@interface class-name
  
@implementation class-name
  
@end
```
[+][-](return-type) method-name [ [ (formal-parameter) ] ];
- (void) method1: (int) x;
- (int) method2: (int) x second: (float) y;
- (int) method3: (int) x, (float) y;

[obj-name method-name];
[myObject add1: 1];
[myAdder add2: 7: 5: 3];
- (int) method2: (int) x second: (float) y;
[myObject method; 7 second: 3: 2];
Adder *myAdder = {Adder: allow1[],};

// The getter for sum
- (int) sum();
return sum();

// The setter for sum
- (void) setSum: (int) a {
  sum = a;
}

@property int sum;

psyObject setSum: 100;
newSum = myObject sum);
myObject sum = 100;
newSum = myObject sum;

// stack.h - Interface and implementation of a simple stack
import <Foundation/Foundation.h>

// Interface section
@interface Stack: NSObject
@public
int stackArray [100];
int stackTop;
int stackMax;
int stackMin;
@end

- (void) push: (int) number;
- (void) pop;
- (int) top;
- (int) empty;
@end
Ruby

- Classes similar to Java
- Dynamic typing
  - Adding members (extension)
- initialize - constructor
- Information Hiding
  - Dynamic access control
    - public, private methods
    - Data members are private
  - Access via getter/setter methods - attributes
- Evaluation
  - Dynamic typing and access control reduces readability/reliability
  - More flexible
class MyClass
  def meth1
    ...
  end

  def meth2
    ...
  end

  private
  def meth3
    ...
  end

  end # of class MyClass

def sum
  sum
  end
  def sum=(new_sum)
    sum = new_sum
    end

Parametric ADTs

- Aka Generic types
  - E.g. stack of any particular type
- Ada
  - Generic packages
  - E.g. generic stack
  - Compile-time instantiation
- C++
  - Template classes
  - Compile-time instantiation
- Java, C#
  - Generic classes
  - Only one physical instantiation
  - Also bounded types

```plaintext
generic
Max_Size : Positive; -- A generic parameter for stack -- size
type Elements_Type is private; -- A generic parameter -- for element type
package Generic_Stack is
  -- The visible entities, or public interface
  type Stack_Type is limited private;
  function Empty(Stk : in Stack_Type) return Boolean;
  procedure Push(Stk : in out Stack_Type;
                  Element : in Elements_Type);
  procedure Pop(Stk : in out Stack_Type);
  function Top(Stk : in Stack_Type) return Elements_Type;
  -- The hidden part
private
  type List_Type is array (1..Max_Size) of Elements_Type;
type Stack_Type is record
    List : List_Type;
    Toppos : Positive range 0..Max_Size - 1;
    End;
end Stack_Type;
end Generic_Stack;
```

```plaintext
template instantiation // Type is the template parameter
class Stack { 
private:
  type elements_type;
  int top; 
public:
  // A constructor for 100 element stack 
  Stack() { 
    elements_type* stack = new type [100];
    elements_type* top = 0;
    // A constructor for a given number of elements 
    Stack(int size) {
      elements_type* stack = new type [size];
      elements_type* top = 0;
    }
  } // A destructor
  void delete elements_type[]; // A destructor
  void push(elements_type new_value); // Push element onto list
  void pop(); // Remove element from list
};
```

```
```
Encapsulation

- Group related data and operations together
  - Unit for decomposition
  - Programmer can concentrate on items in unit
  - Separate compilation
  - Libraries
- C
  - Files provide encapsulation and separate compilation
  - Header files allow definitions to be referenced in other files
    - #include is textual copy, can physically copy as well
    - Programmer takes responsibility for header file contents and use
- **C++**
  - Files as in C
  - Class headers
    - Operations on multiple classes problematic
    - E.g., multiplication of Matrix & Vector
    - Friend functions
- **Ada**
  - Package
    - Can contain multiple declarations
- **C#, Java**
  - Assembly, JAR
  - Created by tools
  - Primarily for building libraries

---

```cpp
class Matrix, //** A class declaration
class Vector {
  friend Vector multiply(const Matrix, const Vector);
  ...}
}
class Matrix { //** The class definition
  friend Vector multiply(const Matrix, const Vector);
};
//** The function that uses both Matrix and Vector objects
Vector multiply(const Matrix m1, const Vector v1) {
  ...}
```

---

**Naming Encapsulations**

- Limit scope of names
  - Large multi-programmer projects
  - Libraries
- **C++ namespace**
  - Code enclosed in namespace block
  - Multiple files can use same namespace
  - Scope resolution operator ::
    - using and using namespace
- **Java package**
  - File declared as part of package
  - Unnamed package
  - Scope qualifiers (public, private, protected, package scope)
  - Qualified names
  - import
Ada package
- Also namespace control
- Visibility: with
- Qualified names
- using

Ruby module
- Collections of methods and constants
  "Methods include module name"
- Visibility: require