ABSTRACT DATA TYPES

- Based on the fundamental concept of ABSTRACTION:
  - process abstraction
  - data abstraction

- Both provide:
  - information hiding
  - reliability
  - security
  - reuse
ADT PROPERTIES

- Defines a collection of objects, and
- a Set of applicable operations

- Representation of objects is hidden
- Operations by outsiders is restricted to only those operations that are visible
COMMON FORMAT OF DESCRIPTION

- **SPECIFICATION**
  - Defines type name and parameters
  - Names visible operations and results types

- **BODY**
  - Describes syntax of type objects
  - Describes visible and hidden operations
IMPLEMENTATIONS

• SIMULA 67 - first to introduce classes, retrospectively recognized to be ADTs
• CLU - an experimental language that introduced ADTs
• Modula-2 - first generally accessible implementations in a widely used language
• Smalltalk - used ADTs as basis for objects
• Ada - used ADTs in packages
• C++ - added ADTs to C
ENCAPSULATION and INSTANTIATION

● ENCAPSULATION
  - the syntax of the specification as a separate module
  - builds a “fire-wall” around the type
  - provides a reusable, portable object
  - the development of the idea of a type as an object

● INSTANTIATION
  - the creation of an instance of the type
  - the operation of importing an ADT into another program unit
  - may include initialization actions
  - scope may be limited to the lifetime of the user module or may be specified
Abstract Data Types -
details

- Type representation and operations on that type are defined together.

- Representation is hidden from user of the type -- objects of type t can only be manipulated by operations defined for t.

- Advantages of user-defined ADTs
  - encapsulation
  - protection
  - extensibility

- We’ll look at three languages:
  - Simula 67
  - Ada
  - Modula-2
Simula 67: Classes

- A class consists of:
  - variable declarations
  - procedure declarations
  - code (for initialization)

- If C is a class with variables $x_1...x_n$ and procedures $p_1...p_k$, an instance of C is a dynamically created object, say r.

  \[
  \text{ref (C) r;}
  \]

  \[
  \text{...}
  \]

  \[
  \text{r :- new C;}
  \]

  \[
  \text{...}
  \]

  \[
  \text{...r.x_i...}
  \]

  \[
  \text{...r. p_j(y_1 ... y_m)...}
  \]
Stack Example

class stack;
    begin
        integer array a(1..100);
        integer top;

        boolean procedure empty;
            ...
        end;

        procedure push (element);
            ...
        end;

        procedure pop;
            ...
        end;

        procedure look;
            ...
        end;

        top := 0;
    end stack;
Using the Stack Class

ref (stack) s1,s2;

...

s1 :- new stack;
s2 :- new stack;
s1.pop; -- error
s1.push(5);

...

s1.look; -- 5

...

s2.look ; -- error

...

But no protection!

s2.a(4) := 1000;  allowed, but unsafe.
s1.top := 0;
Inheritance in Simula

- If $x$ is a subclass of $y$, then instances of $x$ have all of $x$'s attributes plus all of $y$'s attributes.
  - $x$ inherits the attributes of $y$.

- Example: defining a heterogeneous stack

  ```simula
class stack_mem
  begin ref(stack_mem) next_mem
  next_mem :- none
  end stack_mem;
  ```
Example Continued: Define stack

class stack;
    begin
        ref (stack_mem) first;
        ref (stack_mem) procedure top
            top :- first;
        procedure pop;
            if not(empty) then
                first :- first.next_mem;
        boolean procedure empty;
            empty :- (first = = none);
        procedure push(e);
            ref(stack_mem) e;
            begin
                if first =/= none then
                    e.next_mem :- first;
                    first :- e;
                end
            first :- none;
    end stack;
Example Continued: Stackable Objects

- Stackable objects must be instances of a *subclass* of stack_mem:

  stack_mem class complex(...) -- declare complex as subclass of stack_mem

  ...

  end complex

- Another example:

  class mammal;

  mammal class dog;

  mammal class cat;

  dog class golden_retriever;
Packages in Ada

- **Two parts:**
  - specification: provides interface, defines visibility.
  - body: provides implementation

- **Important:**
  - Support separate compilation so that if package p1 uses package p2, p1 can be compiled given only the specification part of p2.
package stack is  -- the specification
  type stacktype;
  function empty (s: in stacktype)
      return boolean;
  procedure push (e: in integer;
      s: in out stacktype);
  procedure pop (s: in out stacktype);
  function top(s: in stacktype)
      return integer;
end stack;

package body stack is  -- the body
  type stacktype is . . .
  function empty (. . .) is . . .
  ...

Package example (continued)

- Does our separate compilation rule hold:
  - No!
  - Definition for stacktype must be in the interface too.

- Problem: We didn't want stacktype's definition to be exported.
  - Solution: Divide the specification into a *public* part and a *private* part.
package stack is -- the visible part
    type stacktype is private;
    function empty(...) ...
    procedure push ...
...
private -- the private part
    type list_type is array (1..100) of int;
    type stacktype is
        record
            list : list_type;
            top : integer range 0..100 := 0
        end record;
    end stack;
Using Packages

with stack;

procedure p is
    s : stack.stacktype;
    begin
    ..
    stack.push(4,s);
    ...stack.top(s)...
    ...
    end
OR...

with  stack; use stack;

procedure p is
    s : stacktype;
    begin
    ...
    push(4,s);
    ...top(s)...
    ...
    end
Ada Generic -- Abstract Package

generic
  -- A private generic type means assignment and equality must be
  -- defined on that type
  type Elem is private ;

package List is
  type T is private ;
  -- Create operation is implicit. Lists created by declaration
  procedure Head (L: T ; V: out Elem ; Err: out ERROR_INDICATOR) ;
  -- Length can’t fail so no need for error indicator
  function Length (L: T) return NATURAL ;
  procedure Tail (L: T ; LT: out T; Err: out ERROR_INDICATOR ) ;
  -- Cons can’t fail so no need for error indicator
  function Cons (L: T ; V: Elem ) return T ;

private
  -- an Ada access type corresponds to a Pascal pointer
  -- the entity referenced by the pointer is defined in the package body
  -- In this case, it would be a record with one field pointing to the next
  -- list element
  type LISTREC ;
  type T is access LISTREC ;

end List ;
Modules in Modula-2

- Very similar to Ada packages, but only pointer types can be exported.

  Definition module stack; -- public
  type stacktype;
  procedure empty . . .
  ...
  end stack;

  Implementation module stack; -- private
  type stacktype = pointer to record
  list: . . .
  topsub: . . .
What are the repercussions of this design decision?

- separate compilation is easy (+)
- module must supply a creation/initialization routine (-)
- extra use of pointers (-)