G22.2210

Programming Languages: PL

B. Mishra
New York University.

Lecture # 9
November 6 1995

—Slide 1—

Data Abstraction
Encapsulation and Inheritance

- Specification of the *Abstract Data Types*
  (Design of attributes and operation)
- Implementation of *Concrete Data Types*
- Encapsulation — Usage does not depend on implementation details.
- Mechanisms
  - *Subprograms*
  - *Type Definitions*
  - *Inheritance*
- **History**
  - (Pascal 1970) Type Definition: defines structure of a data object with its possible value binding.
  - Type can be then bound (as an attribute) to a variable name
  - (Ada, Modula, etc.) Further extended to
    set of data objects + *set of operations*

—Slide 2—

Data Abstraction and Encapsulation of Programmer Defined Data Types

Abstract Data Types

- *Set of Data Objects* (uses other data type definitions)
- *Set of Abstract Operations* on those data objects
- *Encapsulation of the Whole* in such a way that the user of the new type cannot manipulate data objects except by the user defined operations.

1) **Package** 2) **Module** 3) **Class**
4) **Flavor** 5) **Form** 6) **Structure**
7) **Cluster**

- Distinction between

  Ada Packages — Partition the static program text

  C++ Class — Also describes dynamic objects (at compile time)

—Slide 3—

Stack Example in Ada

```
package STACK_PACKAGE is
  procedure PUSH(C: in Character);

  procedure POP;

  function TOP return Character;

  function EMPTY return Boolean;

  function FULL return Boolean;

  procedure INIT;
end STACK_PACKAGE;

package body STACK_PACKAGE is
  MAX_LEN: constant Integer := 1000;

  TOP: Integer := -1
  ELEMENTS: array (0..MAX_LEN -1) of Character;
    ...
```

—Slide 4—

STACK_PACKAGE *Continued*

```
  procedure PUSH(C: in Character) is
    begin if TOP < MAX_LEN then
          TOP := TOP + 1; ELEMENTS(TOP) := C;
    end if; end PUSH;

  procedure POP is
    begin if TOP > -1 then
          TOP := TOP - 1;
    end if; end POP;

  function TOP return Character is
    begin return ELEMENTS(TOP); end;

  function EMPTY return Boolean is
    begin return TOP = -1; end;

  function FULL return Boolean is
    begin return TOP = MAX_LEN - 1; end;

  procedure INIT is begin null; end;
end STACK_PACKAGE;
```

—Slide 5—

STACK *in C++*

```
#define MAX_LEN 1000

struct STACK {
  char ELEMENTS[MAX_LEN];

  int TOP = -1;
  enum {EMPTY = -1; FULL = MAX_LEN -1};

  void PUSH(char C){TOP++; ELEMENTS[TOP] = C;}

  void POP(){TOP--;}

  char TOP(){return (ELEMENT[TOP]);}

  boolean EMPTY(){return (boolean)(TOP == EMPTY);}

  boolean FULL(){return (boolean)(TOP == FULL);}

  void INIT(){}
};
```

—Slide 6—

Abstraction Facilities

Set of Modules:

Each module performs a limited set of operations on a limited amount of data.

- Information is encapsulated. Implementation hiding. Representation independence.
  1. User *does not need to know* the hidden information to use the abstraction.
  2. User *is not permitted* to directly use or manipulate hidden information.
- Ada, C++, Smalltalk provide language support for data encapsulation.
- In Ada, the declaration "`is private`" for type makes the internal structure inaccessible. Only subprograms within the package have access to "`private`" data.

—Slide 7—

Modules & Classes

- **Modules**:
  - *Static*
    – cannot create new modules or copies of existing modules at run-time
  - Interface (*Public View*)
    – Collection of declarations: Types, Variables, Procedures, etc.
  - Implementation (*Private View*)
    – Code for the procedures ... may be changed without affecting the interface.
- **Classes**:
  - *Dynamic*
    – Class of objects may be created and deleted at run time
  - *Constructor & Destructor*
    – Constructor contains initialization process
    – Destructor contains clean-up and "last-rites" (will-and-testament) process

—Slide 8—

Example in C++

```
const int max_len = 255;

class string {
  char *s;
  int len;                             \\PRIVATE

public:                                \\PUBLIC form here
  string(){s = new char[max_len]; len = max_len - 1;}
  string(char *p){len = strlen(p); s = new char[len+1];
                  strcpy(s,p);}        \\CONSTRUCTORS

  ~string() {delete s;}                \\DESTRUCTOR

  void operator = (string&);
  void operator += (string&);
  string operator + (string&);

  void assign(char* st){strcpy(s,st); len=strlen(st);}
  void print(){cout << s << "\nLength" << len << "\n";}
};
```

—Slide 9—

string *continued*

```
void string::operator = (string& a)
{strcpy(s,a.s); len = a.len;}

void string::operator += (string& a)
{strcpy(s+len,a.s); len += a.len;}

string string::operator + (string& a)
{
  string temp(len+a.len);
  strcpy(temp.s,s);
  strcpy(temp.s+len, a.s);

  return(temp);
}
```

—Slide 10—

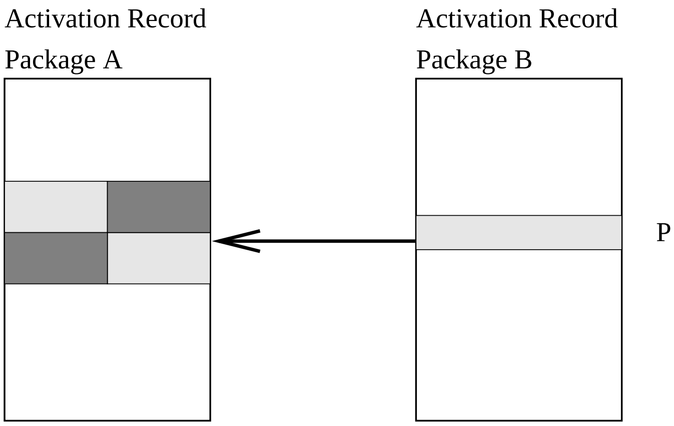
string *Usage*

```
main()
{
  string one("Bud_Why_Err"),
         two("Why ask Why"),
         both, four;

  both = one;
  both.print();
  both += two;
  both.print();
  both = both + both;
  both.print();
  four.assign("This Bud´s for you");
  both = four + two;
  both.print();
}
```

—Slide 11—

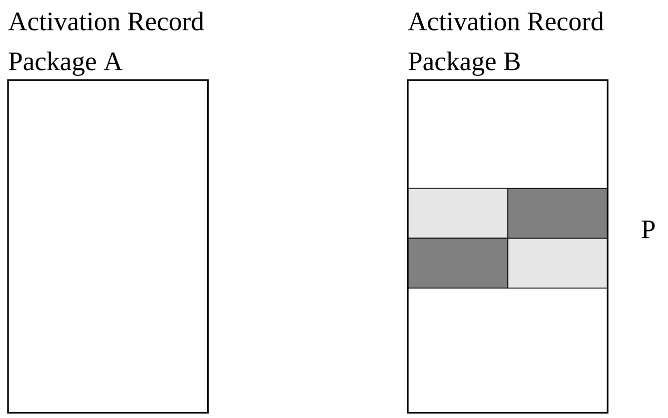
Implementation Issues
Indirect Encapsulation



- The structure of the abstract data type is defined by the package specification A
- Actual storage is maintained in an activation record for package A
- In package B, which declares and uses the objet P, the run-time activation record contains a pointer to the actual data storage.
- Huge run-time cost — Low recompilation cost

—Slide 12—

Implementation Issues
Direct Encapsulation



- The structure of the abstract data object is defined by the specification for package A
- The actual storage for object P is maintained within the activation record for package B
- High run-time execution efficiency — High compile-time cost. Use of an abstract data type requires the compiler to know the details of representation (including the private components)
- Ada uses *direct encapsulation*

—Slide 13—

Direct vs. Indirect Encapsulation in Ada

Indirect

```
package A is
  type MyStack is private;
  procedure NewStack(S: out MyStack);
  ...
  private
  type MyStackRep;   -- Hidden

  type MyStack is access MyStack Rep;
end;
```

Direct

```
package A is
  type MyStack is private;
  procedure NewStack(S: out MyStack);
  ...
  private
  type MyStack is record   -- also hidden
    TOP: integer;
    A: array (1..100) of integer;
  end record;
end;
```

—Slide 14—

Generic

A generic abstract data type definition

allows certain attribute of the type to be specified separately so that one base type definition may be given, with the attributes as parameters, and then several specialized types derived from the same base type may be created.

- Like type definitions with parameters
- Parameters may be type names as well as values
- Parameters may affect the operations in the abstract data type definition

Generic Facility in Ada

- Provides means of parameterizing subprograms and packages
- Generic packages provide families of abstract data types
- Generic subprograms provide families of abstract operations

—Slide 15—

Use of `generic` *in Ada*

```
package QUEUE_PACKAGE is
  procedure APPEND(C: in Character);

  procedure REMOVE(C: out Character);

  function IS_EMPTY return Boolean;

  function IS_FULL return Boolean;

  procedure INIT_QUEUE;

  procedure DESTROY_QUEUE;
end;
```

—Slide 16—

Generic QUEUE_PACKAGE

```
generic
  type Elem is private;
  MAXELTS: in Natural;
package QUEUE_PACKAGE is
  type Queue is limited private;

  procedure APPEND(Q: in out Queue; C: in Elem);
  procedure REMOVE(Q: in out Queue; C: out Elem);
  function IS_EMPTY(Q: in Queue) return Boolean;
  function IS_FULL(Q: in Queue) return Boolean;
  procedure INIT_QUEUE(Q: in out Queue);
  procedure DESTROY_QUEUE(Q: in out Queue);

  FULL_QUEUE, EMPTY_QUEUE: exceptions;
private
  type Queue is
    record
      FIRSTELT, LASTELT: Integer range 0..MAXELTS - 1 := 0;
      CURSIZE : Integer range 0..MAXELTS := 0;
      ELEMENTS: array (0..MAXELTS - 1) of Elem;
    end record;
end QUEUE_PACKAGE;
```

—Slide 17—

Generic Formal Parameters

can be: *Object parameters*, *Type Parameters*, or *Subprogram parameters*

- **Object Parameters**
  – Similar to parameters of subprograms
  – Modes: `in` & `out`

```
generic
  MAX_LEN : in Natural; --Local Constants
```

- **Type Parameters**
  – Necessary to specify some of the properties of the actual type parameters to allow syntactic and semantic checks to be performed on the body of the generic subprogram independently of the generic instantiation.

```
generic
  type Item is private;
  type Sort_Array is array (positive range <>) of Item;
   with function "<"(u,v: Item) return Boolean is <>;
function HEAP_SORT(A: Sort_Array) return Sort_Array is
   ...
end HEAP_SORT;
```

—Slide 18—

Generic Subprogram Parameters

Three Forms:

- `with function SUM(X, Y: Item) return Item;`
  – An actual function of the appropriate function type

  $$\texttt{Item} \times \texttt{Item} \rightarrow \texttt{Item}$$

  must be provided.

- `with function "*"(X, Y: Item) return Item is <>;`
  – An actual function may be omitted
  – The default is obtained from the instantiation.

- `type Enum is (<>);`

  `with function NEXT(A: Enum) return Enum is Enum´Succ;`

  – An actual function may be omitted

  – The default is then the one specified in the declaration: (e.g., `Enum´Succ`)

—Slide 19—

Inheritance

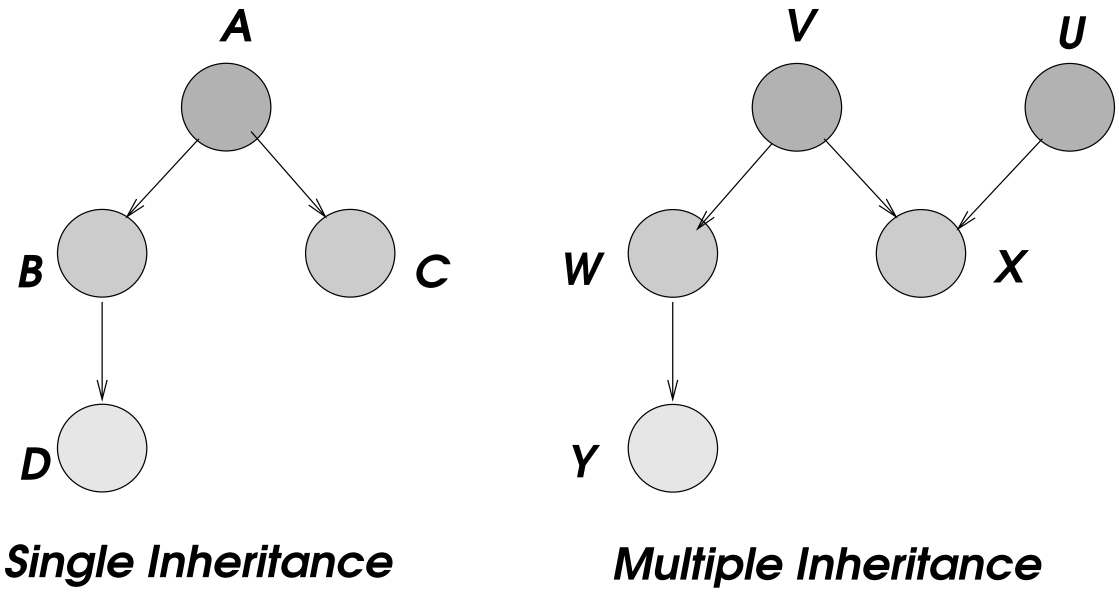
Information passed implicitly among program components

- **Inheritance is**
  the receiving in one program component of properties or characteristics of another component *according to the special relationship* that exists between the two components.
  - Passing of data and functions between independent modules.
  - $A \Rightarrow B$ — inheritance relationship between class A and class B is established.
  - If a certain object X is declared within class A and is not redefined in class B, then a reference to X in B actually refers to object X of class A via *inheritance*.

—Slide 20—

Single & Multiple Inheritance

- $A \Rightarrow B$
  - A is the *parent* class, *base* class or *superclass*
  - B is the *child* class, *dependent* class or *subclass*



- *Single Inheritance*: Class can only have a single parent.
- *Multiple Inheritance*: Class can have multiple parents.

—Slide 21—

Example: C++

```
class elem {
public:
  elem(){v = 0}
  void ToElem(int b){v = b;}
  int FromElem(){ return v;}
private:
  int v;}

class ElemStack: elem {
public:
  ElemStack() { size=0;}
  void push(elem i)
    {storage[++size] = i;}
  void pop()
    {return storage[size--];}
  void MyType() {printf("I am type ElemStack\n")}
protected:
  int size;
  elem storage[100];}
```

—Slide 22—

Derived Class and Method Inheritance

```
class NewStack: ElemStack {
public:
  int peek(){ return storage[size].FromElem()}
}

. . .

{ elem x;
  ElemStack y;
  int i;
  read(i);
  x.ToElem(i);   // Coercion to elem type
  y.push(x);
  ...
}
```

—Slide 23—

Mixin Inheritance

- $A \Rightarrow B$ — Class B is derived from and is a modification of class A.
- *Mixin Inheritance*: Define only the difference between the base class and the new derived class.

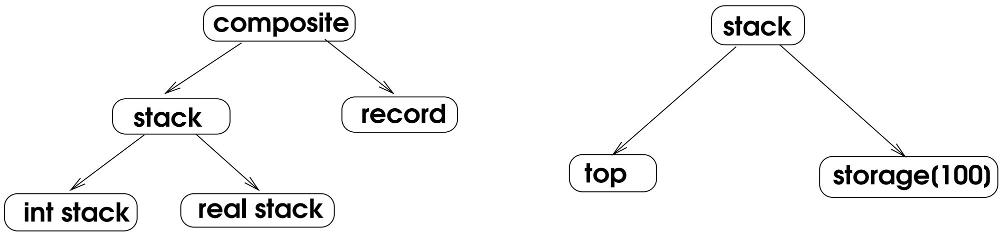
```
deltaclass StackMod{
  int peek(){return storage[size].FromElem();}
}// Does not exist in C++

class NewStack
  = class ElemStack + deltaclass StackMod;
```

—Slide 24—

Abstraction Concepts
Specialization & Decomposition

If $A \Rightarrow B$ means B is a related class to A
what is the relationship between objects of A and B?



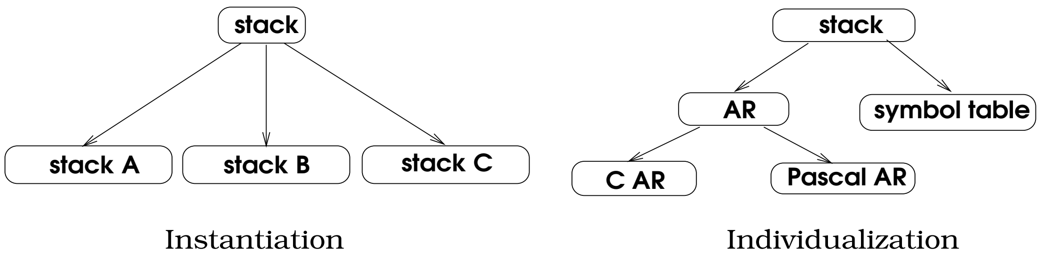
Specialization Decomposition

- **Specialization**
  – Allows the derived object B to obtain more precise properties than A.
  – Class `NewStack` is a specialization of `ElemStack`
  – *Converse process,* **Generalization**.

- **Decomposition**
  – Separates an abstraction into its components
  – *Converse process,* **Aggregation**.

—Slide 25—

Abstraction Concepts
Instantiation & Individualization



- **Instantiation**
  - Creates instances of a class (copy operation)
  - *Converse process,* **Classification**.
- **Individualization**
  - Groups similar objects with common purposes
  - *Converse process,* **Grouping**.

—Slide 26—

Polymorphism

Polymorphism means the ability of a single operator or a subprogram name to refer to any number of functions, depending on the data types of the arguments and results.

- Usually, parameters are assumed to have L-values. Polymorphism allows one to go beyond simply those objects with L-values.
- Note `+` in `1 + 2` and `1.1 + 2.1` is *overloaded*, signifying polymorphism.
- Examples: `generic` in Ada and `template` in C++ allow limited forms of polymorphism

—Slide 27—

Polymorphism in ML

ML and Smalltalk allow for the most general form of polymorphism

```
      fun ident(x) = x;
----> val ident = fn: ´a->´a

      ident(3)         ---->   val it=3 : int
      ident("abc")     ---->   val it="abc" : string
      ident([1,2,3])   ---->   val it=[1,2,3] : int list
```

- *Difference between ML polymorphism and Ada generic*
  - Ada compiles separate function instances, one for each type
  - ML uses a single instance.

—Last Slide—

Functional Polymorphism

ML polymorphic functions with function arguments

```
fun length(nil)=     0
    | length(a::y)=  1+length(y);

fun printit(nil)=    print("\n")
    | printit(a::y)= (print(a); printit(y));


fun process(f, l)=   f l;

process(printit, [1,2,3]);
----> 123

process(length, [1,2,3]);
----> 3
```

[End of Lecture #9]