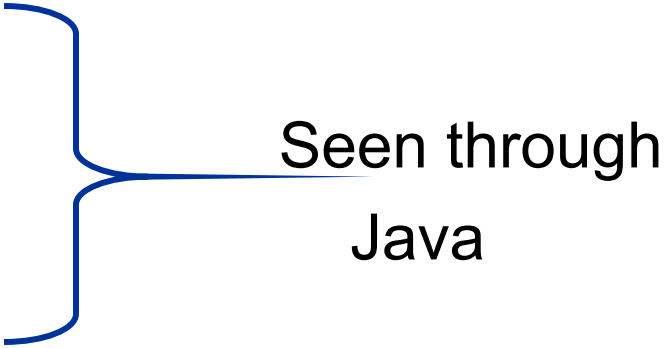
Object Orientation Introduction

Based on the slides of Maurizio Gabbrielli

Outline

- Data abstraction (chapter 9)
- Abstract Data Types: Overview
- Object-Oriented Programming
  - Methodology
  - Linguistic concepts
    - Encapsulation
    - Subtypes
    - Inheritance
    - Dynamic Lookup

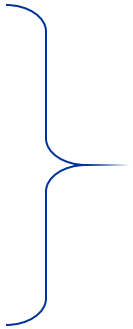


Control abstraction

- Subprograms, blocks, parameters

```
double P (int x) {
    double z;
    /* Function body
    return expr;
}
```

- Specify P
- Write P
- Use P



without knowing the context

Data abstraction

- Data type = values + operations

  `Integer = [-Maxint..Maxint]` And `{+,-, *, Div, mod}`

  - Operations are the only way to manipulate an integer
- Data abstraction: the representation of data and operations is inaccessible to the user

What language support for abstraction?

- Control abstraction
  - Hide realization in the body of procedures
- Data abstraction
  - Hide decisions about the representation of data structures and the implementation of operations
  - Example: A priority queue made by
    - A binary search tree
    - A partially ordered vector

- Which linguistic support is provided by a language for this Information hiding?

Abstract Data Types (ADT)

- One of the major contributions to the languages of the years ' 70
- Basic Idea:
  - Separates the interface from the implementation
    - Example:
      - **`Sets`** have **`Empty`**, **`Insert`**, **`Union`**, **`Is_member?`**, ...
      - **`Sets`** implemented as... vector, concatenated list...
  - Use type control to ensure separation
    - The developer has access only to the operations of the interface
    - The implementation is *encapsulated*

Primitive types VS abstract types

- Example: int
  - Declare variables x: int
  - Set of Operations +,-, *,...
  - No other operation applicable to integers

- Similar properties for abstract types
  - Declare variables x: abstract_type
  - Define a set of operations (interface)
  - The language ensures that only these operations are applicable to the values of abstract_type

Encapsulation principle

- Representation independence

> Two correct implementations of a (abstract) type are not distinguishable from users of that type

  - Implementations are editable without interfering with any user
  - Because the user can not access the implementation

Reality or ideal?

- In Clu, ML,... the independence from representation is a *theorem*
  - Demonstrable because the language restricts access to implementation: only through the interface

- In C, C++, it's an ideal
  - The independence of representation is supported by "*Good programming practices*"
  - Language does not guarantee it

    Example: access to bit representation of -1

Modules

- General construct for the *Information hiding*
- Two parts
  - Interface:

    A collection of names and their types
  - Implementation:

    Declarations (of types and functions) for each interface name

    Additional hidden declarations (to the user)

- Examples:
  - Modula modules, Ada packages, ML structures,...

Data forms and abstraction

```
module Set
   interface
         type set
         val empty : set
         fun insert : elt * set -> set
         fun union : set * set -> set
         fun isMember : elt * set -> bool
   implementation
         type set = elt list
         val empty = nil
         fun insert(x, elts) = ...
         fun union(…) = ...
         ...
end Set
```

Modules can be used to define ADT

Some languages separate interface from implementation

An interface can have multiple implementations

Abstraction and Modularity

- **Component**
  - Programming Unit
    - function, data structure, module
- **Interface**
  - Types and operations defined in a component that are visible outside of the component
- **Specification**
  - "Intended semantics" of the component, expressed through properties observable through the interface
- **Implementation**
  - Data structures and functions defined in the component, not necessarily visible from outside

Example

- Component
  - function that calculate the square root
- Interface
  - float sqroot (float x)
- Specification
  - if x > 1 then sqrt (x) * sqrt (x) = x.
- Implementation

```
float sqroot (float x){
   float y = x/2; float step=x/4; int i;
   for (i=0; i<20; i++){if ((y*y)<x) y=y+step;
                        else y=y-step; step = step/2;}
   return y;
}
```

Example 2

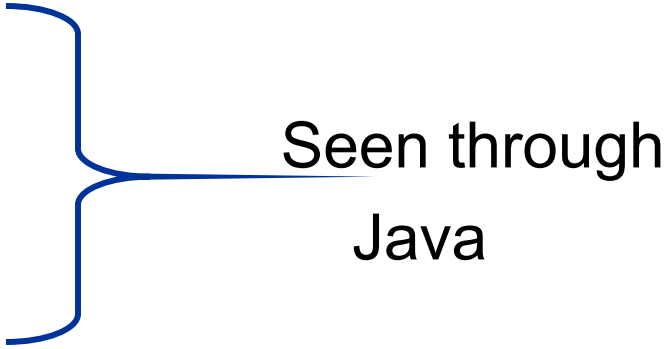
- Component
  - Priority queue: a data structure that returns items in descending order of priority
- Interface
  - Type `Prioqueue`
  - Operations

```
empty     : Prioqueue
insert    : Elemtype * Prioqueue → Prioqueue
deletemax : Prioqueue → Elemtype * Prioqueue
```

- Specification
  - `insert` adds an item to the collection of stored items
  - `deletemax` returns the element with highest priority and the queue of the remaining elements
- Implementation ...

Outline

- Data abstraction
- Abstract Data Types: Overview
- **Object-Oriented Programming**
  - Methodology
  - Linguistic concepts
    - Encapsulation
    - Subtypes
    - Inheritance
    - Dynamic Lookup



Object-Oriented languages

- Information hiding and encapsulation: primitive language concepts
- In an extensible context
  - That allows re-use of the code (more on it later)
  - Easy programming
    - close to our mental abstraction
- Languages:
    - Historical: Simula, Smalltalk
    - Commercial: C++, C#, Java, ...

Objects and classes

A Object is a box that contains

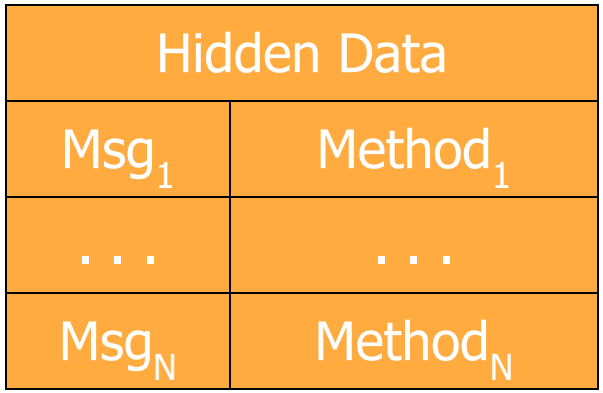
- Hidden Data: variables, values; even functions
- Public Operations: Methods;

An Object Oriented Program

- Send messages to objects
- An object responds to msg
- State confined in objects

Principles of organization allow to group objects having the same structure (e.g., classes). The same principles allow Extendibility And Reuse.

Abstraction on data and on Control, information hiding and encapsulation are present since from the beginning



Object-Oriented Programming

The O-O programming is simultaneously:

- A programming methodology
  - To organize concepts using objects and classes
  - To build extensible systems

- A series of linguistic concepts
  - *Encapsulation* of data and functions in objects
  - Subtypes allow *the extension* of data types
  - Inheritance allows the *reuse* of implementations