

CSE 307: Principles of Programming Languages

Modules and Encapsulation

R. Sekar

Topics

1. Abstraction

Abstract Data Type
Modules

Section 1

Abstraction

Abstraction

- Objective of every programming language
 - managing program complexity
- Primary means for complexity reduction
 - Abstraction
- We abstract often-used “computation patterns” by more compact equivalents.

4 / 14

Abstraction (Continued)

- We can trace the use of abstractions from early days of computers:
 - represent programs using bit-patterns, as opposed to “rewiring” circuits
 - replace hard-to-remember machine instructions by assembly instructions.
 - abstract repeated patterns in assembly instructions by macros
 - allow direct expression of higher level concepts such as compound types, loops, and functions into programs.

5 / 14

Motivation

- Primitive types:
 - insulate programmers from implementation details
 - e.g., representation of floating point numbers
 - provided with a set of operations that have “expected” behavior
- Compound types
 - operations provided only to access/modify fields
 - implementation details are visible throughout program
- ADT (Abstract Data Type)
 - hide implementation details
 - provide set of meaningful operations as with primitive types

6 / 14

ADT

- Type is characterized by a set of operations
- Encapsulation: Only way to access the data is through these operations
 - access to internal representation of ADT is restricted
- Information hiding:
 - Semantics of operations don't depend on implementation
 - implementation can be changed without affecting "client code", i.e., code that uses this ADT
- Supports following design goals
 - modifiability/maintainability, reusability, security

7 / 14

Algebraic Specification of ADT

- type complex imports real;
- operations:
 - $+$: $\text{complex} \times \text{complex} \rightarrow \text{complex}$
 - $-$: $\text{complex} \times \text{complex} \rightarrow \text{complex}$
 - $*$: $\text{complex} \times \text{complex} \rightarrow \text{complex}$
 - $/$: $\text{complex} \times \text{complex} \rightarrow \text{complex}$
- makecomplex : $\text{real} \times \text{real} \rightarrow \text{complex}$
- realpart : $\text{complex} \rightarrow \text{real}$
- imagpart : $\text{complex} \rightarrow \text{real}$

8 / 14

Algebraic Specification of ADT (Contd.)

- axioms
 - $\text{realpart}(\text{makecomplex}(r,s)) = r$
 - $\text{imagpart}(\text{makecomplex}(r,s)) = s$
 - $\text{realpart}(x+y) = \text{realpart}(x) + \text{realpart}(y)$
 - $\text{imagpart}(x+y) = \text{imagpart}(x) + \text{imagpart}(y)$
 - $\text{realpart}(x-y) = \text{realpart}(x) - \text{realpart}(y)$
 - $\text{imagpart}(x-y) = \text{imagpart}(x) - \text{imagpart}(y)$
 -

9 / 14

ADT in Standard ML

```

abstype 'element Queue = Q of 'element list
with
  val createQ = Q [];
  fun enqueue (Q l, e) = Q (l @ e);
  fun dequeue (Q l) = Q (tl l);
  fun frontq (Q l) = hd l;
  fun emptyq (Q []) = true
    | emptyq (Q h::t) = false;
end;
type 'a Queue
val createq = -: 'a Queue

```

10 / 14

Modules

- More general than ADTs
 - a way to group “semantically related” code that may or may not operate on a single type
- Program unit with a public interface and private implementation
 - May include private operations
- Export datatypes, variables, constants, functions
- Ideal to support
 - separate compilation
 - library facilities
 - namespace separation (to avoid name clashes)

11 / 14

Java Packages

- A package is a group of related classes
- Classes in other packages referenced using a qualified name <pkg>.<name>
- “import” keyword can be used to reduce clutter due to qualified names
- Other related features
 - relationship between file names and class names
 - no need for separate header files

12 / 14

Modules in C

- C does not support modules
 - Functionality partially simulated using files
- Namespace pollution can be managed using “static” keyword
 - name visible only in the current file
 - overloaded meaning - static in some contexts means static memory allocation
- “extern” keyword used in a file to declare symbols to be located in other files
 - interface exported by a module can be specified in a corresponding header file
 - this header file “#include”d by users of this module
- linker deals with name resolution across files

13 / 14

C++ Name spaces

- Name spaces can be declared as follows:

```
namespace <name> {  
    <declarations and/or functions>  
}
```
- A name Y within a namespace X can be accessed using a qualified name X::Y
- A “using” declaration can be used to import all names within a namespace

14 / 14