§6.5-6.8: Writing Your Own Library

•devo

17 Oct 2005
CMPT14x
Dr. Sean Ho
Trinity Western University

Reminders:

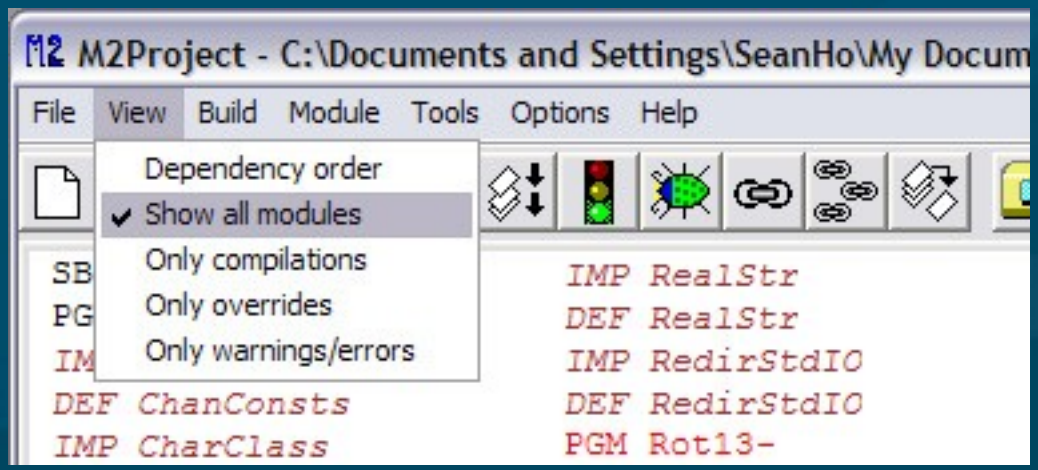
- *journals in folder*
- *no quiz today*

Library modules vs. programs

- Library modules (e.g. STextIO) are different from program modules (e.g. HelloWorld):
  - Linker needs to know what procedures / entities are available for import from a library
  - Programs that use a library don't need to know the implementation of its procedures
- Hence there are two parts to a library module:
  - DEFINITION (a.k.a. header file)
  - IMPLEMENTATION (a.k.a. code file)

Viewing standard library modules

- You can see the definition and implementation files for standard library modules in Stonybrook:
  - View -> Show all modules
  - Each library has a DEF and an IMP
- Make sure not to modify the standard libraries!



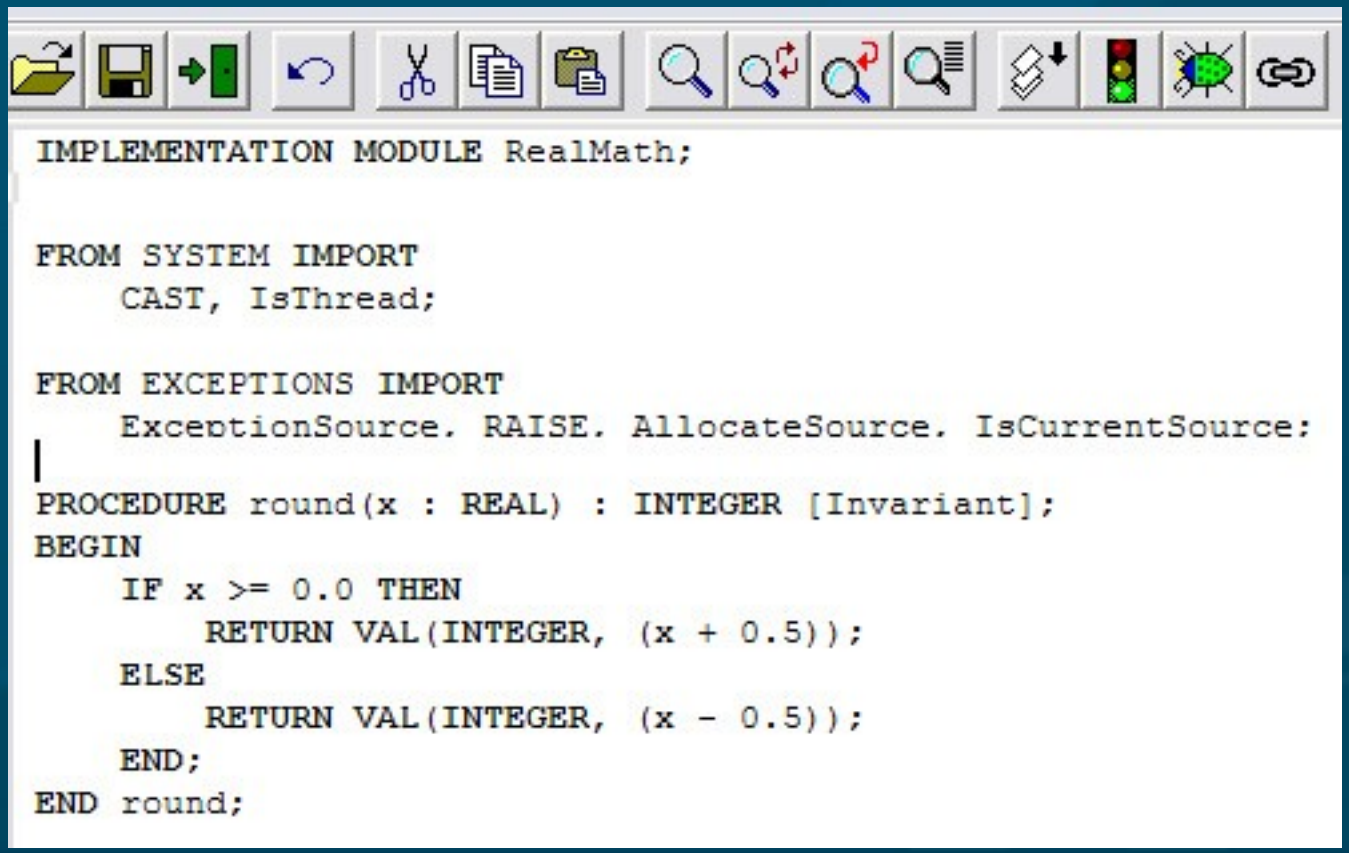
An example of a definition file

- Let's peek at RealMath's DEF:
- Keyword DEFINITION
- No bodies to the procedures
- No body to the module
- No BEGIN anywhere

```
DEFINITION MODULE RealMath;
(* ========================================
            Definition Module from
                  ISO Modula-2
Draft Standard CD10515 by JTC1/SC22/WG13
    Language and Module designs © 1992 by
BSI, D.J. Andrews, B.J. Cornelius, R. B. Henry
R. Sutcliffe, D.P. Ward, and M. Woodman

========================================*)

CONST
    pi          = 3.14159265358979323846;
    expi        = 2.71828182845904523536;

%IF InlineFpp %THEN
PROCEDURE sqrt(x : REAL) : REAL [FppPrim, Invariant];
%ELSE
PROCEDURE sqrt(x : REAL) : REAL [Invariant];
%END

%IF InlineFpp %AND IA32 %THEN

PROCEDURE exp(x : REAL) : REAL [FppPrim, Invariant];

PROCEDURE ln(x : REAL) : REAL [FppPrim, Invariant];
```

An example of an implementation

- Here's part of RealMath's IMP:
- Keyword IMPLEMENTATION
- Parameter lists must match DEFINITION
- IMPORTs as needed
- Body of module optional (initialization)



```
IMPLEMENTATION MODULE RealMath;


FROM SYSTEM IMPORT
    CAST, IsThread;

FROM EXCEPTIONS IMPORT
    ExceptionSource, RAISE, AllocateSource, IsCurrentSource;

PROCEDURE round(x : REAL) : INTEGER [Invariant];
BEGIN
    IF x >= 0.0 THEN
        RETURN VAL(INTEGER, (x + 0.5));
    ELSE
        RETURN VAL(INTEGER, (x - 0.5));
    END;
END round;
```

Module interface

- The interface of a module is the way in which other modules use it: its publicly accessible
  - Procedures, variables, types, constants, etc.
- There may be other procedures, variables, etc. that are internal to the module and should be hidden from public view
- When designing a module, think carefully about its public interface
  - c.f. preconditions

Example: Fractions ADT

- Often modules are used to define abstract data types: let's make a Fraction type:

```
DEFINITION MODULE Fractions;
```

- We can represent a fraction a/b internally as an ordered pair (array of length 2) of integers:

```
TYPE Fraction = ARRAY [1 .. 2] OF INTEGER;
```

- This Fractions module will contain the Fraction type as well as all the procedures we need to use variables of type Fraction

Creating a library function: Inv

- Let's make a function that inverts a fraction:
  - In the definition module:

```
PROCEDURE Inv (x : Fraction) : Fraction;
```

  - In the implementation module:

```
PROCEDURE Inv (x : Fraction) : Fraction;
VAR temp : INTEGER;
BEGIN
    temp := x[1];
    x[1] := x[2];
    x[2] := temp;
END Inv;
```

Using our Fractions library

- Let's try to use our new Fractions library:

```
MODULE FractionTest;
FROM Fractions IMPORT
    Inv;
VAR
    applesPerFriend, friendsPerApple : Fraction;
BEGIN

    friendsPerApple := Inv (applesPerFriend);
END FractionTest.
```

- Oops, forgot to provide a way to initialize a Fraction!

Hiding implementation details

- Since a Fraction is just an ARRAY [1..2] OF INTEGER, so FractionTest could initialize just by

```
(* initialize with 1.5 apples per friend *)
applesPerFriend[1] := 3;
applesPerFriend[2] := 2;
```

- But we want the Fractions library to hide the fact that a Fraction is really an array
  - Array is just an implementation detail
  - Future version could use some other way
  - Public interface should stay the same
  - A user could set denominator to zero

Accessor functions

- We can provide an accessor function that allows the user to get at the numerator and denominator in a controlled manner
  - Error checking (denominator ≠ 0)
  - Consistent interface (if we switch from arrays)
- Set function:
  - Assign (num, denom : INTEGER) : Fraction
- Get functions:
  - Numerator (x : Fraction) : INTEGER
  - Denominator (x : Fraction) : INTEGER

Set/Get functions

- In the definition module:

```
PROCEDURE Assign (num, denom:INTEGER): Fraction;
```

- In the implementation module:

```
PROCEDURE Assign (num,denom:INTEGER): Fraction;
VAR x : Fraction;
BEGIN
    x[1] := num;
    x[2] := denom;
    RETURN x;
END Assign;
```

  - Add error checking:

Design choices for error handling

- No error checking:
  - Make sure preconditions are clearly stated in documentation: writeup, comments, user manual, WriteString, etc.
  - Not ideal – users can be very creatively bad!
- Precondition checking:
  - Use IF to check preconditions in code
  - If bad input, take evasive action
- Postcondition checking:
  - ReadResult: if error, continue but set a flag

TODO items

- Lab5 due today/tomorrow/Wed:
  - §6.11 #(25 / 33) (choose one)
- Quiz ch6 on Wed
- Homework due Fri: 6.11 #28
- Quiz ch7 on Fri
- Reading: through §7.5 for Wed