

Lecture 24 Encapsulation and ADTs

Encapsulation

- hides complex data and methods

Step 1: Visibility Modifiers to Hide Information

- we can declare *methods* or *variables* in the class
 - public
 - private
 - called *visibility modifiers*

public

- any method (even from other classes) can
 - access public variables
 - invoke public methods

private

- restricts access to a variable or method
- only methods in *this class can access*

```
class Rational {
    private int numerator, denominator;
}
```

- numerator **cannot** be accessed by classes that use Rational such as Lab5
- numerator can **only** be accessed by methods in Rational
- any Rational object can access numerator of any other Rational object
- Math.PI, Math.pow() are public, any class can use
- cannot see the private ones in Math class, no examples
- private variables and methods are for use within the class only, hidden from others — we never publish them!

Step 2: Accessor and Mutator Methods to Get/Set Info

- *accessor methods*
 - method to *get* the value of a private variable
- *mutator methods*
 - method to *set* the value of a private variable(s)

```
class Rational
{
    // accessor methods
    public int getNumerator()
    { return numerator; }
    public int getDenominator()
    { return demoninator; }

    // mutator methods
    public void setNumerator( int num )
    { numerator = num; reduce(); }
    public void setDenominator( int den )
    { denominator = den; reduce(); }
    public void setNumDen( int num, int den )
    { numerator = num;
      denominator = den; reduce(); }
}
```

- provides us with superb control
- we are **HIDING** the **INSTANCE VARIABLES**
- access to private instance variables is guarded by methods
 - can refuse setting to insane values
 - eg: may refuse setting denominator = 0
 - can modify other instance variables that may depend on this one
 - eg call reduce()

Abstraction

- shows simplification only, hides details using encapsulation
- we never need to know how a Rational number is built
- define all required operations: plus, minus, ...
- fully encapsulated the `numerator`, `denominator`
 - private variables and methods can do “dirty work”
- now think of a *new data type* Rational
 - forget about numerator, denominator details
 - just the concept matters
 - call this an *abstraction*
 - eg: `String` class
 - we understand only the public methods
 - forget how string of chars are stored
 - forget how to multiply Rational, it just happens
- you think abstractly
- you hide details
- programming is tedious, unnatural
- abstraction in programming is natural

Abstract Data Types

An *Abstract Data Type (ADT)* hides the implementation details of a collection of data behind a well-defined interface. An ADT implements one principle concept, eg Complex numbers, without revealing how the data is structured.

The *interface* is the set of public methods used to operate on the data in the ADT. Objects give us a convenient way to construct an ADT.

By hiding the details, but providing methods to access the entity, we are performing abstraction. This is good programming practise, because it allows us to stay at arm’s length away from messy details.

Similar term:

- API or *application programming interface* — a published interface for others to use

All good software systems begin with good, well-documented interfaces!

Example ADT: String

- data: contains a bunch of characters
 - how? don’t care.... array of characters?
- `.substr()` method
- `.charAt()` method
- `.compareTo()` method
- etc

Sample ADT: List

example: a list of book titles in a library

- define operations you want
- a **constructor**, to create a new list
- **append** a new book to end of the list
- **print** the entire book list
- **get title** of fourth book in list
- is a given book title in the list?
- **insert** new book *before* another book in list
- **delete** a book from list
- **important for library**: preserve order of books in the list
- the ADT is defined by the above abstract operations
- decide how to implement the list as a program
- encapsulate the ADT in a class
- a list of books becomes an object
- ADT must remember a list of book titles
 - one book title: a String
 - multiple books: a String[]
 - how many books? need a counter numBooks

1. Define the Interface

- for each method, choose identifier, parameters, return type

```
class Library {
    public Library()
    public void append(String newTitle)
    public void delete(String title)
    public void printBooks()
    public String getTitle(int bookNum)
    public boolean isInList(String title)
    public void insertBefore(String newTitle,
        String titleAfter)
}
```

2. Expand the Interface

- plan how each method works
 - group related methods together
 - how much work does each do?
 - if a method seems complex
 - complex: need additional methods to help,
 - eg:


```
private int find(String title)
• used for isInList(), delete(), insert()
```
 - write pseudocode, and comments for each method


```
// this method inserts a new book before
//
public void insert( ... )
```
- incremental coding and testing: fill in stub code
 - stub code does no work
 - compiles cleanly
 - returns some sane ‘dummy value’ if needed
 - really useful for testing, write 1 method at a time
 - replace it with real code later

3. Implement and Test the Interface

- decision: use an array of Strings for all books in the library
- note: the ADT was already defined before choosing an array

```

private int    numBooks;
private String[] books;

public Library()
{
    books = new String[100];
    // reserve space for 100 books (bad style)
    numBooks = 0;
}

public void append( String newTitle )
{
    books[numBooks] = newTitle;
    numBooks++;
}

public void printBooks()
{
    for( int i=0; i < numBooks; i++ ) {
        System.out.println( books[i] );
    }
}

public String getTitle( int bookNum )
{
    if( 0 <= bookNum && bookNum < numBooks )
        return books[bookNumber];
    else
        return "Error, no such book";
}

```

```

public boolean isInList( String title )
{
    int position = find( title );
    // convenient to add private find method
    return( position >= 0 );
}

// return position if found, -1 if not found
private int find( String title )
{
    // no binary search: array is not sorted
    for( int i=0; i < numBooks; i++ ) {
        if( books[i].equals(title) )
            return i;
    }
    return -1;
}

public void delete( String title )
{
    int position = find( title );
    if( position < 0 ) return;
    // must copy all books from position+1
    // back one position
    numBooks--;
    for( int i=position; i < numBooks; i++ ) {
        books[i] = books[i+1];
    }
}

```

```

public void insertBefore( String newTitle,
String titleAfter )
{
    int position = find( titleAfter );
    if( position < 0 ) {
        // not found, add new book at end of list
        append( newTitle );
    } else {
        // move all books forward one position
        // makes room at 'position' for newTitle
        for( int i=numBooks; i > position; i-- )
            books[i] = books[i-1];
        books[position] = newTitle;
        numBooks++;
    }
}
}

```

- example was a list of book titles
- array of String objects
- can be list of anything, say an Account objects
- limitations of this list
 - only allows 100 books
 - when array fills? code is broken!
 - can you think of way to fix this?
 - insert(), delete()
 - must move many items in array
 - worst case, must move almost all elements in array
 - these methods are $O(n)$
 - not very efficient, there are faster ways
 - can you think of way to do this in $O(1)$?
- next lecture: how to make a better list!