CS211, LECTURE 16
GENERIC PROGRAMMING

ANNOUNCEMENTS:

READING:

DS&SD: Chap 6, App. B

DS&A: Chap 5

OVERVIEW:

Motivation

Abstract Procedure

Abstract Data Type

Search/Sequence View

Collections API

1. Motivation

- ❒ Good Software Design
  - information hiding
  - resuse
  - abstraction
- ❒ Information Hiding
  - debugging
  - prevent inconsistencies
- ❒ Reuse
  - extensibility
  - time saving
- ❒ Abstraction
  - Procedures
  - Data

2. Procedural Abstraction

- ❒ Stubbing
  - Method header explains role of method
  - Write method headers first
  - Develop method bodies later
- ❒ Procedural Abstraction
  - Determine problem's tasks and subtasks
  - Write method stubs
  - Develop method bodies later
- ❒ Advantages
  - Hides details
  - Postpone details for later
  - Allows for improvements later
  - Allows for early tests of program
  - Reuse of methods ("plug-n-play")

3. Data Abstraction

- ❒ Type
  - classification
  - helps with programming
- ❒ Data and Types
  - ***data type***: set of values and operations on them
  - Think of **`class`** (classification)
  - **`class`** has information and action
- ❒ Implementation of type
  - **`interface`** can define type
  - **`class`** can implement interface in different ways
- ❒ Data Abstraction
  - separate data type from implementation
- ❒ Advantages
  - OOP!

4. Types of Types

- ❒ atomic/primitive
  - not built from other types
  - numeric, Boolean, character, enumerated
- ❒ enumerated
  - finite set of values
  - **`enum`** in C/C++ (not in Java: use **`interface`**)
- ❒ aggregate
  - collections of data elements
  - arrays, sequences, records
- ❒ array (fixed size, elements same type)
- ❒ sequence (dynamic array)
- ❒ record
  - fields with mixed types (**`struct`**)
  - class is an extension of this

5. Abstract Data Type

- ❒ ADT Definition:
  - "*A set of data values and associated operations that are precisely specified independent of any particular implementation.*" (http://www.nist.gov/dads/)
  - Gist: ADT = set of values & set of operations
- ❒ Again, Why?
  - data abstraction in programming
  - mathematical theory! can derive useful things without worrying about specific language
- ❒ Three approaches to developing
  - axiomatic
  - constructive
  - postcondition
- ❒ First, a brief example…

6. Brief Example

- ❒ See http://www.nist.gov/dads/HTML/abstractDataType.html
- ❒ Stack (S), value (v)
- ❒ Methods:
  - new() returns a stack
  - push(v,S) pushes a value on top of the stack
  - pop(S) pops a value off the top of the stack
- ❒ Specifications in books differ…
- ❒ List ADT? Array ADT? Tree ADT?

7. Axiomatic Approach

- ❒ Rather theoretical…
- ❒ The gist:
  - syntax elements (what the ADT does)
  - semantics elements (how the ADT does "it")
- ❒ Syntax
  - name
  - sets
  - signatures
- ❒ Semantics
  - preconditions
  - build operations
  - axioms

8. Constructive Approach

- More theory…
- The gist:
  - express operations of an ADT in terms of operations of another ADT
  - the "another ADT" is already defined and forms underlying model for new ADT
- Example:

```
class MyArray {

    private final int size;
    private Object[] a;
    public MyArray(Object[] a) {
        size = a.length;
        this.a = a;
    }

    public void print() {
        for(int i=0;i<size;i++)
            System.out.print(a[i]+" ");
        System.out.println();
        }

    }
}
```

9. Postcondition Approach

- A bit less theory…
- The gist:
  - specify semantics of each operation in terms of preconditions and postconditions
- Precondition
  - specify relationship(s) that must exist among the input(s) for the function to be defined
  - what must be true for a function before it runs
- Postcondition
  - what must be true after a function runs
- Postcondition approach useful for developing classes

10. ADTs in Java

- ADT Syntax
  - ADT: name of type, involved sets, method signatures (signatures contain the sets)
  - Java: define abstract class or interface
- ADT Semantics
  - ADT: give pre- and postconditions for method
  - use postcondition approach
  - see DS&SD, 6.4.2 for example
- ADT Implementation
  - select representation and formulate algorithms
  - specification (interface) is implemented by class
  - abstract → concrete

10.1 Example 1

```
interface Printable {
    void print();
}

abstract class Matrix2D implements Printable {
    protected final int size1;
    protected final int size2;
    protected Object[][] a;
    public Matrix2D(Object[][] a) {
        size1 = a.length;
        size2 = a[0].length;
        this.a = a;
    }
    abstract public void print();
}

class MyMatrix2D extends Matrix2D
        implements Printable {

    public MyMatrix2D( Object[][] a ) {
        super(a);
    }

    public void print() {
        for(int i=0;i<size1;i++) {
            for(int j=0;j<size2;j++)
                System.out.print(a[i][j]+" ");
            System.out.println();
        }
        System.out.println();
    }
}
```

```
public class TestADT {

    public static void main(String[] args) {

    Printable m = new MyMatrix2D(new Integer[][]
        {
            {   new Integer(1), new Integer(2),
                new Integer(3) },
            {   new Integer(4), new Integer(5),
                new Integer(6) }
        } );

    m.print();

    }
}

/* Output:
    1 2 3
    4 5 6
*/
```

10.2 Example 2

```
interface IPrimitive {
    String toString();
    Object toObject();
}

class Primitive implements IPrimitive {
    private Object v;
    public Primitive(int v) { this.v=new Integer(v); }
    public Primitive(boolean v) {this.v=new Boolean(v);}
    public String toString() { return v.toString(); }
    public Object toObject() { return v; }
}

public class TestPrimitive {
    public static void main(String[] args) {
        IPrimitive i1 = new Primitive(1);
        IPrimitive i2 = new Primitive(true);
        IPrimitive[] p = {i1,i2};

        for (int i=0;i<p.length;i++)
            System.out.println(p[i]);

        for (int i=0;i<p.length;i++) {
            Object v = p[i].toObject();

            if (v instanceof Integer) {
                int x = 1 + ((Integer) v).intValue();
                System.out.println(x); }

            else if (v instanceof Boolean) {
                boolean y = false &&
                        ( (Boolean) v ).booleanValue();
                System.out.println(y); }

        } // end for
    } // Method main
} // Class TestPrimitive
```

11. Search/Sequence View

- ❒ Graph: “tree with loops”
- ❒ NPuzzle example has multiple states
- ❒ Possible algorithms to solve

11.1 Search/Sequence Structure

- ❒ toDo set: ordered nodes to visit
- ❒ sequence structure:
  - need to obtain nodes in a certain order
  - toDo set can grow and shrink
  - need stacks, queues, priority queues, heaps
  - fast lookup unimportant
- ❒ done set: visited nodes
- ❒ search structure:
  - need to organize nodes to search collection
  - search trees, hash tables
  - fast lookup is important

11.2 Sequence/Search Structure Interfaces

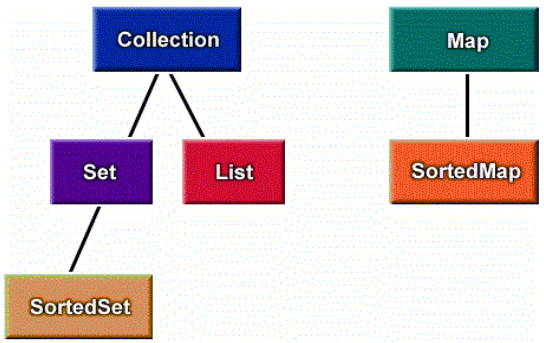
```
public interface SequenceStructure {

    //  Inserts comparable object obj into the structure:
        void put(Object obj);

    //  Extracts the biggest object from the structure:
        Object get();

    //  Checks whether the structure is empty:
        boolean isEmpty();

    //  Returns the number of items stored in
    //  the structure:
        int size();
}

public interface SearchStructure {

    //  Put object o into search structure:
        void insert(Object o);

    // Remove all objects equal to o from
    //  search structure:
        void delete(Object o);

    //  Return result of search for object o:
        boolean search(Object o);

    //  Return number of elements:
        int size();
}
```

12. Collections API

- ❒ **`java.lang`**:
  - automatically imported
  - **`Object`**, **`Cloneable`**, **`Comparable`**, wrappers, and more!
  - http://java.sun.com/j2se/1.4/docs/api/java/lang/package-summary.html
- ❒ **`java.util`**:
  - builtin data structures and algorithms!
  - need to import these classes
    ex) **`import java.util.*;`**
    ex) **`import java.util.Arrays;`**
  - http://java.sun.com/j2se/1.4/docs/api/java/util/package-summary.html
- ❒ Tutorial:
  - Lookfor ***Collections API***
  - http://java.sun.com/docs/books/tutorial/collections/index.html

13. Overall View of Collections

- ❒ *Interface* hierarchy (from Java Tutorial):



- ❒ Also, see App. B in DS&SD!
- ❒ The gist:
  - ***collection***: generic data structure
  - ***set***: collection with no duplicates
  - ***list***: ordered collection
  - ***map***: collection with key-value associations

14. Example of Collections API

```
import java.util.*;

public class MyList {
    public static void main(String[] args) {

        LinkedList list = new LinkedList();

        list.add(new Integer(1));
        list.add(new Integer(2));
        list.add(new Integer(3));
        list.add(new Integer(4));

        for ( Iterator it = list.iterator();
                    it.hasNext(); )
            System.out.print(it.next()+" ");

        list.remove(new Integer(2));
        System.out.println();

        for ( Iterator it = list.iterator();
                    it.hasNext(); )
            System.out.print(it.next()+" ");

        System.out.println();
    }
}

/* Output:
    1 2 3 4
    1 3 4
*/
```

15. Suggested Exercises

- Write a general class for a singly-linked list.
- Write a **`myArray`** class as a wrapper for any array. The class will "know" about an array's size and supply a method for printing the "internal" array's contents. Define operations **`put(index)`** and **`get(index)`** for the array.
- Add more primitive types to "Example 2". Is it possible to write a generic **`toValue()`** method that would work for all primitive types?
- Create your own list class by extending the **`AbstractList`** class in the the Collections API. Add a **`display`** method.
- Create a class **`ListAsArray`** that implements basic list operations using an array.