Abstract Data Types: Collections, Stacks

I had code for managing a collection of CDs, and I needed code for managing a collection of Shapes…

I know CDs are not Shapes, but they are both collections! How can I reuse the code?

Abstract Data Types

- An abstract data type (ADT) is a set of data and the particular operations that are allowed on that data.
- Data Type is really about techniques managing collections of data in certain ways.
- Abstract means the operations you can perform on it are separated from the underlying implementation.

For every collection we examine, we should consider:
- How does the collection operate, conceptually?
- What operations are included in the interface to the collection?
- What kinds of problems does the collection help us solve?
- How might the collection be implemented?
- How do the implementations compare from an efficiency point of view?
Collections

- A collection is an object that serves as a repository for other objects
- A collection provides services to add, remove, and manage its elements
- The underlying data structure used to implement the collection is independent of the operations provided
- Collections can be separated into two categories
  - linear: elements are organized in a straight line
  - nonlinear: elements are organized in something other than a straight line
- Ordering of elements, relative to each other, is usually determined by either
  - the order in which they were added to the collection
  - or some inherent relationship among the elements

Stacks and Queues as Collections

- A stack
  - Last-in, first-out (LIFO) property
    - The last item placed on the stack will be the first item removed
  - Analogy
    - A stack of dishes in a cafeteria
- vs: A queue
  - First in, first out (FIFO) property
    - The first item added is the first item to be removed
    - Analogy
      - A queue of train commuters
Stack Animation

Adding an element
Removing an element

Top of stack

“Contract” for the Stack Collection

Stack operations

- **Create** an empty stack
- **Add** a new item to the stack
- **Remove** from the stack the item that was added most recently
- **Retrieve** (but not remove) from the stack the item that was added most recently
- Determine whether a stack is **empty**
- Find out **how many** items are in the stack

<table>
<thead>
<tr>
<th>Stack ADT operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>push</td>
<td>Adds an element to the top of the stack</td>
</tr>
<tr>
<td>pop</td>
<td>Removes an element from the top of the stack</td>
</tr>
<tr>
<td>peek</td>
<td>Examines the element at the top of the stack</td>
</tr>
<tr>
<td>isEmpty</td>
<td>Determines if the stack is empty</td>
</tr>
<tr>
<td>size</td>
<td>Determines the number of elements on the stack</td>
</tr>
</tbody>
</table>
**public interface** Stack\<E\>  
{
    /** Adds an item to the top of a stack. */
    public void push(E newItem);

    /** Removes the top of a stack. */
    public E pop();

    /** Retrieves the top of a stack. */
    public E peek();

    /** Determines whether stack is empty. */
    public boolean isEmpty();

    /** Determines whether stack is empty. */
    public int size();
}

---

**The need for Generic Types <E>**

- Assume we have defined a **Group** class that stores and manages a group of objects
- **Group** could store **Objects** (which can hold any type – thanks to polymorphism!)
- But it’s no longer clear to the compiler what methods are available for the objects in my group…

```java
(Die) (myGroup.getMember()).roll();  // Loss of control, and awkward. What if it’s not a Die object?
```

- Instead, a **generic type Group** will be able to **store, operate on, and manage** objects **whose type is not specified until the class is instantiated**

```java
public class Group <E>{
    //definition
}
```
## Generic Types `<E>`

- Instantiating a Group of Product objects
  ```java
  Group<Product> group1 = new Group<Product>;
  ```

- Instantiating a Group of Friend objects
  ```java
  Group<Friend> group2 = new Group<Friend>;
  ```

- You **cannot** instantiate a generic type `<E>
  ```java
  Group<E> bad_group = new Group<E>;
  ```

- You can be more specific: We want to store `Comparable` items
  ```java
class Group<E extends Comparable<E> {
    // declarations and code that manages objects of type E
  }
  ```

## Using a Stack – push and pop

```java
public class StackTest {
  public static void main (String[] args) {
    Stack<String> stk = new Stack<String>();
    stk.push("one");
    stk.push("two");
    stk.pop();
    stk.push("three");
    System.out.println("Contents of Stack: " + stk);
  }
}
```

// What does stk contain now?

How can we print all the elements of a stack without destroying it?
Print a Stack without destroying it
/** @returns String with the contents of stk from top to bottom,
* assuming that the E on the stack have own toString() method */

public String toString (Stack<E> stk) {
    // Create a temporary stack to hold contents of stk
    Stack<E> tempStack = new Stack<E>();
    String s = "[";

    while( !stk.isEmpty() ) {
        E element = stk.pop();
        s = s + element.toString() + " ";
        tempStack.push(element);
    }
    s = s + "]";
    // restore contents of stk
    while( !tempStack.isEmpty() )
        stk.push(tempStack.pop());
    return s;
}

Example: Checking for Balanced Braces
(the first thing a compiler is doing)

An example of balanced braces
a{b[c]{d[e]}f}g

Examples of unbalanced braces
a{b} : Too many closing braces
c{d}e : Too few closing braces
{f{g}h} : Mismatching braces
Checking Balanced Braces: Helper Methods

/* returns true if c is an open bracket */
public boolean open_bracket (char c) {
    return (c == '(') || (c == '{') || (c == '[') || (c == '<');
}

/* returns true if c is a close bracket */
public boolean close_bracket (char c) {
    return (c == ')') || (c == '}') || (c == ']') || (c == '>');
}

/* returns the closing bracket matching the input open bracket */
public char matching_bracket (char c) {
    if (c == '(') return ')';
    else if (c == '{') return '}'
    else if (c == '[') return ']
    else return '>';
}

Checking Balanced Braces Pseudocode

while (still more chars to read {

    get next char in the string

    if it is open_bracket
        then push it on top of the stack

    if it is a close_bracket
        pop char off stack
        check to see if it matches bracket

}
Checking Balanced Braces

Pseudocode

Start by declaring input string balanced
while (still more chars to read && string balanced) {

    get next char in the string

    if it is open_bracket
        then push it on top of the stack

    if it is a close_bracket
        if stack empty => not balanced
            pop char off stack
            check to see if it matches bracket
            if not matched => not balanced

    } if stack not empty => not balanced

/** @returns true if string S has balanced open and closed brackets */
public boolean isBalanced (String s) {
    Stack <Character> stk = new Stack <Character>();
    int i = 0; char nextChar, top; boolean balanced = true;

    while (balanced && (i < s.length())) {
        nextChar = s.charAt(i);  // get the next character in string
        if (open_bracket(nextChar))  // push open brackets onto stack
            stk.push(new Character(nextChar));
        else if (close_bracket(nextChar)) {
            // check whether the matching open bracket is on top of stack
            if (stk.isEmpty()) balanced = false;
            else {
                top = stk.pop().charValue();
                if (nextChar != matching_bracket(top)) balanced = false;
            } // else
        } // else if
        i++;
    } // while
    return (balanced && stk.empty());
} // isBalanced
Implementing a Stack

Implementations of the ADT Stack

- The ADT stack can be implemented using
  - An array
  - A reference-based list
  - The ADT LinkedList
  - The ADT Vector

- Stack Interface
  - Provides a common specification for the three implementations

- StackException
  - Used by Stack Interface
  - Extends java.lang.RuntimeException
package javafoundations;

public interface Stack<E> {

    /**  Adds the specified element to the top of the stack. */
    public void push(E newItem) {
    
    /**  Removes and returns the top element from the stack. */
    public E pop();

    /**  Returns a reference to top element of this stack without
    removing it.*/
    public E peek();

    /**  Returns true if the stack contains no elements and false
    otherwise. */
    public boolean isEmpty();

    /**  Returns the number of elements in the stack. */
    public int size();

    /**  Returns a string representation of the stack. */
    public String toString();
}

Figure 6.4

Array-Based 
Implementation

• ArrayStack class
  – Implements Stack
  – Private data fields
    • An array of Objects called items
    • The index count
    • Top of stack is at count-1

0 1 2 3 4 5 6 7 ... ... 

Figure 6.4
An array-based implementation
15.5 – The java.util.Stack Class

- While it contains operations similar to a classic stack, it contains other, non-Stack methods
  - java.util.Stack does not implement any Stack interface
  - java.util.Stack provides a search operation that attempts to locate a target element returns its distance from the top of the stack

- java.util.Stack extends the Vector class, which supports direct access to elements at specific indices
  - Vector is an adjustable-size array with methods that sound like Linked List:

```java
Vector<String> example = new Vector<String>();
example.add("bob");
example.add(0,"before");
example.get(0);
System.out.println(example.size());
```
15.5 – The `java.util.Stack` Class

- The `java.util.Stack` class was developed mainly as a convenience.
- Much of the added functionality comes through inheritance and interface implementation.
- A stack is not everything a Vector is, so it is not a proper is-a relationship.
- It also violates the premise of a well-designed collection class.

Example: Ever wonder how the computer keeps track of Control Flow?

- Recall this discussion about control flow.
Important Application: Execution Stack

```java
public int first(int a, int b) {
    int c;
    ...
    a = second(c);
    ...
}
public int second(int f) {
    int g;
    ...
    return third(f, g);
}
public int third(int m, int n) {
    ...
    return n;
}
public static void main(String[] args) {
    int i, j;
    ...
    System.out.println(first(i, j));
    ...
}
```

Consider recursive factorial:
What happens when you call `factorial(500000)`?

### Organization Through Packages

- Chapter 3 introduced the use of packages and the `import` statement to access package contents.
- Packages are used to organize classes by related functionality:
  - `java.io` – classes related to input/output
  - `java.text` – classes related to text processing
  - `java.util` – utility classes
- The book organizes the collection classes into a package called `javafoundations`