Abstract Data Types: Collections, Stacks

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 the elements it contains
• 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

---

**The 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**

<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>**

```java
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();
}
```

---

**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 – polymorphism!)
  - But it’s no longer clear to the compiler what methods are available for the objects in my group...

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

  (Die) `(myGroup.getMember()).roll();` 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
  }
  ```

  Loss of control, and awkward.
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
import java.util.*; // For Java’s Stack class
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

```java
/** @returns String representation of the contents of stk from top to bottom,
  * assuming that the E on the stack have their 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

An example of balanced braces
```
abc|d|e|f|g
```

Examples of unbalanced braces
```
a|b|l| : Too many closing braces
c|d|e| : Too few closing braces
f|g|h|l| : Mismatching braces
```

```
Input string Stack or equivalent braces  Stack empty => balanced
(a|b|l) 1. push |" |
2. push |" |
3. pop |
Stack empty => balanced
(a|b|e) 1. push |" |
2. push |" |
3. pop |
Stack empty => not balanced
(a|b|h) 1. push |" |
2. pop |
Stack empty where last "|" encountered => out balance
```
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 '>';  
}

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
}

/** @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 the string
        if (open_bracket(nextChar))  // push open brackets onto the 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 if (stack not empty => not balanced)
            }  
            }  
            return (balanced && stk.empty());
Implementing a Stack

Note: Up to now we have been using a stack. Let’s see how it works inside.

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

```java
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();
}
```

**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

- **Figure 6.4**
  - An array-based implementation

- **Array-Based Implementation Diagram**
  - Shows a stack with elements A, B, C, D and count 4
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>();
exmaple.add("bob");
exmaple.add(0,"before");
exmaple.get(0);
System.out.println(example.size());
```

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

```java
main
\arrow{down}
dot
\arrow{down}
obj.dot();
\arrow{down}
helpMe();
\arrow{down}
helpMe();
```
Important Application: Execution Stack

```java
public int first(int a, int b) {
    int c;
    a = second(c);
    return c;
}
```

```java
public int second(int f) {
    int g;
    return third(f, g);
}
```

```java
public int third(int m, int n) {
    return n;
}
```

```java
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`

WARNING

PAY ATTENTION OR ELSE...

- THERE ARE 2 TYPES OF PEOPLE.
  THOSE WHO WILL PAY ATTENTION TO THE NEXT 2 SLIDES AND THOSE WHO WILL NOT
- THE FIRST KIND WILL LIVE A HAPPY PRODUCTIVE LIFE AS POWERFUL PROGRAMMERS
- THE SECOND TYPE MAY DROWN IN THEIR OWN TEARS BEFORE DROPPING OUT OF CS
- YOU HAVE BEEN WARNED!

Changing CLASSPATH in Unix or MacOSX

Because we import javafoundations; we need to tell Java where to find this package:

```bash
tm$ echo $CLASSPATH
```

```bash
tm$ CLASSPATH=/Users/tm/:
```

```bash
tm$ export CLASSPATH
```

```bash
tm$ echo $CLASSPATH
```

Or we can compile and run with the full path:

```bash
tm$ javac -cp /Users/tm/ Test.java
```

```bash
tm$ java -cp /Users/tm/ Test
```

But make sure you have implemented all the missing methods!!

Or we can tell BlueJ's Preferences... where to find the parent directory
Using CLASSPATH

- CLASSPATH = an "environment variable"
- CLASSPATH stores the list of directories where the compiler and run-time environment looks for classes not found in the API
- CLASSPATH can be set in the shell or as a parameter during program execution
- When a class is required at compile or run-time, if it is not in the API, the CLASSPATH locations are checked for the needed class.
- `javac -cp C:\javafoundations; myDir\myFile.java`
- The `-cp` option sets the CLASSPATH to look for packages containing needed classes in two locations:
  - First, `C:\javafoundations`
  - Second, the current working directory (`.`)
- Locations are separated with semi-colons (Windows) or colons (Unix)