Implementing Stacks
Basics of Exceptions

OK, so stacks are useful

- Stacks have many uses
  - Arithmetic
  - Language parsing
  - Keeping track of recursion (more in this in a week or so)
- How can stacks be implemented?
  - Using a generic List class
    - Works fine, easy to do
    - May not be as efficient
  - Using an array directly
  - Using a linked list
- Tradeoff between generic and tailored implementations
  - Generic implementation: simple, quick
  - Tailored implementation: often more efficient
Review: methods needed for stacks

- Stacks need six methods
  - Create: make a new stack
  - Push: add an element to the top of the stack
  - Pop: remove an element from the top of the stack
  - Peek: examine the element on the top of the stack
  - PopAll: remove all the elements from the stack
  - IsEmpty: return true if the stack has no elements

- Implement these methods using
  - Methods existing for a list
  - Operations on an array
  - Linked list operations directly

Stack using a (generic) list

```java
public class StackList {
    private List l;
    int size;
    public StackList () {
        l = new List();
        size = 0;
    }
    public void push (Object item) {
        l.insert (item, 0);
        size++;
    }
    public Object pop () {
        Object item = l.index (0);
        l.delete (0);
        size--;
        return (item);
    }
    public Object peek () {
        return (l.index(0));
    }
    public boolean isEmpty () {
        return (size == 0);
    }
    public void popAll () {
        while (!isEmpty()) {
            pop();
        }
    }
}
```
Issue: what about empty lists?

- All this works well if we call pop() with things on the stack
- What if we call pop() on an empty stack?
  - This has no reasonable result!
  - Need to indicate an error somehow
- Solution #1: return a special value
  - Return `null` if there’s an error
  - Problem: always checking for `null`!
  - This approach usually taken in C
- Solution #2: generate an exception

What’s an exception?

- An *exception* is an abnormal condition
  - Null reference dereferenced
  - File not found
  - Stack is empty when pop() called
- Exceptions can be dealt with in two ways
  - Handle exception locally
  - Pass it to the calling method
- Pass to calling method
  - Must declare that method can cause an exception:
    public Object pop() throws StackException {...}
  - Calling method must deal with it now!
How can an exception be “caught”?  

- Often useful to “catch” an exception
  - Deal with the problem
  - Try an alternate way of doing things
- Exceptions can be caught with a “try…catch” block
  - Different exceptions can be caught separately
  - Not all exceptions need be caught
- Exceptions are objects
  - May have methods
  - May carry information about the error condition

```java
try {
mystack.pop();
} catch (StackException e) {
    println ("Empty stack!");
}
```

```java
while (true) {
    try {
        f = new FileReader (name);
        break;
    } catch (IOException e) {
        print ("Enter a new name:");
        // get another name
    }
}
```

Stacks with exceptions

```java
public class StackList {
    private List l;
    int size;
    public StackList () {
        l = new List();
        size = 0;
    }
    public Object peek () {
        if (isEmpty()) {
            throw new StackException ("Stack empty");
        }
        return (l.index(0));
    }
    public void popAll () {
        while (!isEmpty()) {
            pop();
        }
    }
    public void push (Object item) {
        l.insert (item, 0);
        size++;
    }
    public Object pop () throws StackException {
        if (isEmpty()) {
            throw new StackException ("Stack empty");
        }
        Object item = l.index (0);
        l.delete (0);
        size--;
        return (item);
    }
    public boolean isEmpty () {
        return (size == 0);
    }
}
```
Implementing stacks with arrays

```java
public class StackArray {
    private Object arr[];
    int size;
    private final int max = 20;
    public StackList () {
        arr = new Object[max];
        size = 0;
    }
    public Object peek () {
        if (isEmpty()) {
            throw new StackException
                ("Stack empty");
        }
        return (arr[size-1]);
    }
    public void push (Object item)
        throws StackException {
        if (size >= max) {
            throw new StackException
                ("Stack full");
        }
        arr[size++] = item;
    }
    public Object pop ()
        throws StackException {
        if (isEmpty()) {
            throw new StackException
                ("Stack empty");
        }
        return (arr[--size]);
    }
}
```

Issues with arrays for stacks

- Arrays are good for stacks because
  - Pop and push are easy to implement
    - Unlike general lists, only need to insert/delete at end
  - Very space efficient
    - Only require space for object references
    - No need for extra links
  - Fast
    - Some CPUs can do these operations in a single instruction
- Downside of using arrays
  - Stack has a limited size: hard to grow beyond that
  - Entire stack must be allocated even if it’s never used
    - May be inefficient if maximum size is 1000, but stack never exceeds 10 elements
- Arrays for stacks are very common
Implementing stacks with linked lists

```java
class StackArray {
    private StackArrayNode head;
    int size;
    public StackArray () {
        head = null;
        size = 0;
    }
    public Object peek () {
        if (isEmpty()) {
            throw new StackException
            ("Stack empty");
        }
        return (head.val);
    }
    public void push (Object x) {
        head = new StackArrayNode
        (x, head);
        size++;
    }
    public Object pop ()
    throws StackException {
        if (isEmpty()) {
            throw new StackException
            ("Stack empty");
        }
        Object obj = head.val;
        head = head.next;
        return (obj);
    }
    private class StackArrayNode {
        public Object val;
        public StackArrayNode next;
        public StackArrayNode
        (Object x, StackArrayNode n) {
            val = x;
            next = n;
        }
    }
}
```

Issues with using linked lists as stacks

- Easier to do specific implementation rather than using generic linked lists
  - Only need to insert / delete at head
  - No need to move through the list
- Implementation is efficient, but not as efficient as arrays
  - More space per object (next reference)
  - Slower operations
- No preset limit on stack size
Next time: uses & implementation of queues

- Class on Monday will deal with queues
- Andy Pohl will give the lecture
  - I’m making the slides
  - It’s not an optional lecture!
- Reading: Chapter 7 in C&P
- I’m at a conference through Wednesday morning
  - My office hours for Tuesday are cancelled
  - Email response will be slower

More on Assignment #2

- Because of issues with exceptions and file reading, there is now a MyFile class
  - See Web page for details on how to get it
- Two methods in MyFile:
  - static MyFile open (String name): opens name and returns a reference to an object that can be used to access the file
    - Returns null if the file couldn’t be opened
  - String readLine (): reads a line from the file and returns it
    - Returns null if an error occurred (like end of file)
    - May also return an empty string (blank line in the file)
  - IOExceptions handled by MyFile class and not passed to calling methods
- You may write your own code, or use MyFile