STACKS

Abstract Data Types

A "stack" is a kind of list, but also an *abstract data type* or ADT.

An abstract data type can be defined in various ways:

- the idea of a way of organizing information, separately from any implementation.
- a set of possible values, together with the operations that can be performed on them.
- the design of a class, with public methods specified but without method bodies or private methods or instance variable specified.

In Java, what methods are needed for the list ADT?

Publicity rules

Not for the first time, nor for the last:

- Don't make methods public unless you have a good reason.
- Don't make variables public.
- If you've heard of "protected" as an alternative to public and private, don't try to use it as a way out of the public-private choice.
  - We might or might not get to protected later in the course. Don't worry about it if you haven't heard of it. Try not to let its use in the text bother you.

(Why is this slide here, where it doesn't seem to belong?)

Definition of a Stack

Data: A sequence of objects. A stack operates in a "last in, first out", or LIFO, manner. The 'place' where elements go in and come out is considered the "top".

A stack is a kind of list — a list with restricted operations.
(There's a Stack class in the Java API that includes all the standard List operations, but most implementations do restrict stacks compared with lists, and that's what we'll do here.)

Operations:

- **push(o):** Add o to the (top) of the stack.
  Requires: the stack is not full.

- **pop():** Remove and return the top element on the stack.
  Requires: the stack is not empty.

- **isEmpty():** Return whether the stack is empty.

- **isFull():** Return whether the stack can't hold any more items.

Note that this is not written in Java! The description is quite general. For example, it does not restrict:

- what can go in a stack
- what a stack is used for
- how a stack is stored
- how the operations work
- what programming language it might be built in

We have described only what the *user* of a stack needs: its "interface" or "front-end".

This is similar to the interface for a physical device, like a drink machine:

- It has buttons you can push to get a drink.
- It doesn't reveal irrelevant details like how the cans are stored, how change is made, etc.

Uses of Stacks

You might have seen stacks used to keep track of method calls. (More on this shortly.)

Stacks are useful for processing that involves 'interruption': pausing what we're doing, doing something else (which might also be interrupted ...), and then resuming what we paused. We can use a stack to remember enough about what we were doing so that we can resume.

Some examples:

- Checking for balanced parentheses in an expression with multiple types of brackets.

- Evaluating algebraic expressions

A Java interface

We can define the interface for a queue using a Java `interface`. It's like a class, but without any instance variables or method bodies. One use for an interface is to describe an ADT.

(We'll define the instance variables and method bodies *separately*, in a class (or classes) that "implement" the interface.)

```
public interface Stack {
  /** Add o to the top of this Stack.
   * Precondition: the stack is not full.
   * @param o The object to be pushed.
   */
  void push(Object o);

  /** Remove and return the top element of this Stack.
   * Precondition: the stack is not empty.
   * @return the former top element of the stack */
  Object pop();

  /** Return whether this Stack is empty.  */
  boolean isEmpty();

  /** Return whether this Stack can't hold any more items.  */
  boolean isFull();
}
```

All interface members are automatically `public`. According to the Java Language Specification: "It is [. . .] strongly discouraged as a matter of style, to redundantly specify the public modifier for interface methods."

Two Views of an Interface

Here are two ways to think of an `interface`:

- A **guarantee** to the client code that any class that implements the interface definitely has methods with these headers (and maybe other methods too).

- An **obligation** of the implementer, who must write these methods.

Using Stack

We can't create instances of `Stack` (it doesn't have any code that *does* anything).

```
public class Broken {
    public static void main(String[] args) {
        Stack s = new Stack(15);
        for (int i=0; i != 15; i++) {
            s.push(new Integer(i));
        }
    }
}
```

However, we can write code to *use* (an instance of a class implementing) `Stack`.

```
public class Okay {
    public static void fill(Stack s, int n) {
        for (int i=0; i != n; i++) {
            s.push(new Integer(i));
        }
    }
    public static void main(String[] args) {
        // To call fill(), we must construct some
        // kind of Stack, so we need a class that
        // implements Stack.
    }
}
```

Implementing an Interface

In order to implement an `interface`, we must write a class that declares that it implements the interface.

For example, a class that implements the `Stack` interface must begin like this:

```
class ... implements Stack {
```

The class must contain two things:

- Bodies of the methods guaranteed by the interface to exist.
- Instance variables to hold the queue contents (or whatever). They should be `private`.

We are allowed to add other things to our class, but these are our obligations.

Implementing Stack

```
/**
 * A Stack with fixed capacity.
 */
public class ArrayStack implements Stack {

  /** The number of elements in this Stack */
  private int top;

  /** contents[0 .. top-1] contains the elements in this Stack. */
  private Object[] contents;

  // Representation invariant:
  //   top >= 0
  //   If top is 0, the Stack is empty.
  //   If top > 0:
  //     contents[top-1] is the top element in the Stack.
  //     contenst[0 .. top-1] contains the elements in the order they
  //       were inserted, excluding those removed already

  /** An ArrayStack with capacity for n elements. */
  public ArrayStack(int n) {
    contents = new Object[n];
  }
```

(Continued on next slide)

```
  /** Add o to the top of this Stack.
   * Precondition: ! isFull()
   */
  public void push(Object o) {
    contents[top++] = o; // What if top = HOWBIG?
  }

  /** Remove and return the top element of this Stack.
   * Precondition: ! isEmpty()
   */
  public Object pop() {
    return contents[--top]; // What if top = 0?
  }

  /** Return true iff this Stack contains no elements. */
  public boolean isEmpty() {
    return top == 0;
  }

  /** Return true iff this Stack has no space left.
   * This is only relevant in the context of some implementations.
   */
  public boolean isFull() {
    return top >= contents.length;
  }
}
```

Questions

Why is the constructor $O($`capacity`$)$?

Using ArrayStack

We couldn't construct a `Stack`, but we can construct an `ArrayStack`.

Example (compiles and runs):

```
public class AlsoOkay {
    public static void fill(Stack s, int n) {
        for (int i=0; i != n; i++) {
            s.push(new Integer(i));
        }
    }

    public static void main(String[] args) {
        // Could also be declared as ArrayStack.
        Stack s = new ArrayStack(15);
        fill(s, 15);
    }
}
```

Exercises

- Use the `ArrayStack` class to write a program that reads strings (until "quit" is entered) and prints them in reverse order.

- Write a class implementing `Stack`, storing the elements in an `ArrayList`.

- Write a class implementing a `Stack` designed specifically to hold `String`s, using an `ArrayList<String>` so as to take advantage of the facilities in Java 1.5.