Partially Implementing an Interface or Class

Consider our stack datatype, with: push, pop, isFull and isEmpty.

Suppose users often ask for the head (without removing it), and we don't want them to have to push and pop to get it. It would be nice if we could write something like:

```
... Stack {
    void push(Object o);
    Object pop();
    Object head() {
        Object result = pop();
        push(result);
        return result;
    }
    boolean isFull();
    boolean isEmpty();
}
```

and if stack implementers could `extend` / `implement` this `Stack`.

Java allows such things, in an "abstract class".

Abstract Classes

An abstract class is halfway between a class and an interface.

It's declared like a class, but with the keyword `abstract`:

```
public abstract class Stack { ... }
```

But like an interface, it can have unimplemented `public` instance methods. These methods are also marked with the keyword `abstract`:

```
public abstract void push(Object o);
```

Instantiating

- You can't instantiate an abstract class.
- If a descendant does not fill in all the missing method bodies, it must be declared `abstract` too.
- If a descendant fills in all method bodies, it does not need to be abstract.

A Tradeoff

We have given a simple rule: Always use an `interface` to represent an ADT.

But another option is to use an `abstract class`. It's not obvious which choice is better.

Reason to prefer an `interface`:

- A class implementing the interface can still extend something else.

Reason to prefer an `abstract class`:

- Can provide code for an operation defined in terms of other ones. A subclass can still override this code if it wants.

Revised rule: Use an `interface` unless you want to provide default method implementation(s).

Question: Why might a subclass override `head()`?