Inner classes

We have seen that it is good to make instance variables private, so they can’t be seen outside the class that defines them.

The same is true of classes. For example, in Project 1, our StoreItem class was defined in our csc216.project1 package.

But there was no reason for any class other than ListOfItems to know about a StoreItem.

Java allows classes to be nested within other classes. A class defined inside another class is known as an inner class.

Inner classes are useful for at least two reasons:

• The name of the inner class is known only within its scope. Thus, it does not “pollute” the namespace of the package.

• The code of an inner class can refer directly to names from enclosing scopes, including both class and instance variables and methods, and local variables of enclosing blocks.

A Fixed Stack

Here is an example adapted from the Java tutorial documentation, a class FixedStack, which implements a stack, and defines an iterator of elements from the stack, from top to base:

```java
public class FixedStack {
    Object array[];
    int top = 0;
    FixedStack(int fixedSizeLimit) {
        array = new Object[fixedSizeLimit];
    }
    public void push(Object item) {
        array[top++] = item;
    }
    public boolean isEmpty() {
        return top == 0;
    }
    // other stack methods go here...
}
```
The interface `java.util.Iterator` is used to communicate a series of values to a client.

Note that `FixedStack` does not directly implement the `Iterator` interface. Why not?

Thus, a separate `adapter` class is used to return the elements one by one. (An adapter class is a class whose main purpose is to implement an interface.)

Note that the adapter class needs to access the array containing the stack elements.

It can directly refer to instance variables of the stack, since the adapter is defined inside `FixedStack`.
More design concepts

Now that we have seen interfaces, inheritance, and inner classes, we know several ways in which two classes might be associated.

We have also seen a couple of ways in which two objects might be associated:

Let’s take a short quiz. For each pair of classes below, how should they be associated?

- A UI window and a pane within that window.
- A CheckingAccount and a BankAccount.
- A video and the customer that rents it.
- Two entities that provide a lessThan and a max function.
- A StoreItem that is only supposed to be used by a ListOfItems.
- A car and an engine.

Delegation

Like private variables, inner classes are useful in hiding information from the rest of the program that doesn’t need to use it.

Why is it good for this information to be hidden?

Sometimes we want to “hide” functions of a superclass. This might arise if we want to implement our simple stack by reusing code from the Java class library.

Consider the java.util.Stack class. How many operations does it have?

Suppose in a program you want a “pure” stack class—one that can only be manipulated via push(...) and pop().
Why would you want such a class, when Java already gives you that and more?

What is the “simplest” way to get a pure Stack class?

Or you could create Stack class “from scratch.” What’s wrong with doing this?

Another option is to create your own Stack class, but have it include a java.util.Stack.

What is the name for the approach are we using here?

Here’s what this class might look like.

```java
public class MyStack
{
    private java.util.Stack stack;
    public MyStack(){stack = new java.util.Stack();}
    public void push(Object o) { stack.push(o); }
    public Object pop() { return stack.pop(); }
    public object top() { return stack.peek(); }
    public boolean isEmpty(){return stack.empty();}
}
```

Another case in which delegation is useful is where an object needs to “change class” during the program.

- An Applicant becomes a Student.
• A Student becomes an Alumnus.

If Applicant, Student, and Alumnus inherit from a common class Person, then such objects need to change class every few years.

What difficulties would this cause in a program?

Let’s solve this problem using delegation.