CMPSCI 187: Programming With Data Structures

Lecture 13: Java's Stack Class and Some Mistakes
7 October 2011

Java’s Stack Class and Some Mistakes

- Issues in Project #3
- Generics Again, and a Correction
- Constructors and Inheritance
- L&C’s Maze Program
- Java’s Stack Class -- the API
- Java’s Stack Class -- the Source Code

Issues With Project #3

- This is a lecture to correct some mistakes -- yours, mine, and L&C's.
- To start with Project #3 -- here's one of yours. One student implemented the dropout stack by removing element 0 of the array (the bottom), then moving every element of the stack down one space in the array to make room for the new top element.
- What's wrong with this? The input/output behavior is exactly what is specified, but we also specified O(1) time for each operation and this push is O(n) if there are n spaces in the array.
- The whole idea is not to move lots of elements. So when we push into a full array, and have to delete the bottom element, the *bottom position changes*. This is the essential feature of a circular array.

More Issues With Project #3

- Remember that your code *must* conform to the properties we need to test it. Don't use packages, give your .java files the right names (now that they are correct on the web site), and make sure that the method signatures agree with the ones in L&C's `StackADT` interface.

- Your `DropoutStack` class should be structurally very much like L&C's `ArrayStack` class. Both are generic classes that have to create an array of elements of their base type. The constructor for such a class has a particular form that you should copy exactly.

- Your class `EmptyCollectionException` is very simple to define because the only difference between an `EmptyCollectionException` and a `RuntimeException` is the name. All you need in your definition is the declaration line and a zero-parameter constructor with no code.

Generics Again, and a Correction

- We want to use generics in order to get their flexibility, but they can be confusing. L&C try to give the simplest explanation they can to allow us to use them, and I think they mostly succeed if we are willing to trust them and follow their usage. But it's worth looking a little more closely at them.

- Vocabulary helps. The class definition that starts "`public class LinearNode<T>`" is not defining the class LinearNode<T>, but the generic class LinearNode. This generic class defines a family of *types*, such as `LinearNode<Dog>` and `LinearNode<SledDog>` -- one for every existing type. These types can also be the classes of specific objects created with the LinearNode constructor.

- I said last time that we could not have `LinearNode<Dog>` and `LinearNode<SledDog>` in the same block of code -- this was wrong. Two classes cannot have the same name at the same time, but these are two types arising from one class. But it's worth noting that neither extends the other.

How Do Generics Work?

- Remember the basic picture, that the compiler checks that the source code is valid and creates a machine-level program for the interpreter to run.

- The "`<T>`" in "`LinearNode<T>`" is not a type but a **type pointer** or **type variable**. When some other piece of code, outside the generic class definition, refers to `LinearNode<Dog>`, the compiler essentially generates a copy of the generic class definition with all the `<T>`'s replaced by `<Dog>`'s. This lets `LinearNode<Dog>` operate as the type of a variable or the class of an object.

- When the constructor `LinearNode<Dog>( )` is called, the compiler runs the constructor `LinearNode( )`, replacing any occurrences of the type variable `T` in that constructor with the class `Dog`. This can create objects whose class is `LinearNode<Dog>`.

More With Generics

- The type variable of a generic class can be more complicated. If we define a class starting "`public class Kennel<T extends Dog>`", the compiler will allow us to use `Kennel<SledDog>`, for example, but not `Kennel<Horse>`. It will apply the definition only to classes extending `Dog`.

- In `ArrayStack` we saw the complication that arises when we want to define an array of T's and we don't know what type T is. The compiler can't run the constructor without knowing how much space to allocate to each T item. So we define an array of Objects, cast it into the type `T[ ]` to allow it to fit into a variable of type `T[ ]`, and endure a warning because the compiler doesn't know that the cast is safe.

Constructors and Inheritance

- I was asked about a common mistake on Question #6 of Midterm #1. You were asked to extend the class `Dog` to a new class `ShowDog` with two new fields, and write a constructor to set both those fields and the two from `Dog`.

- The correct answer was to start the constructor with `super(newName, newAge)`, a call to the `Dog` constructor. Many people instead said `name = newName; age = newAge;` to set the fields correctly. Why is this wrong?

- A constructor does more than set values for the fields of the object it creates. It has to tell the interpreter to lay out memory to store those fields, and this goes on in a particular way. In order to construct a `ShowDog` object, we have to run constructors for `Object`, `Dog`, and `ShowDog` in turn. Each of these constructors creates room for the fields from its class, whether or not the code sets those fields.

Default Constructors

- The normal thing to do in a constructor for an inherited class is to call the parent class' constructor on the first line with `super`, or to call another constructor for the same class with `this`. If you *don't* do either of those things, the compiler does it for you by calling the *zero-parameter* constructor of the parent.

- In the context of that question, the class `Dog` didn't have a zero-parameter constructor, so without an explicit `super` call, the implicit `super( )` call would have failed and the code would not compile.

- So your general habit should be to figure out what kind of `super` call you want and make it. It's also a good habit to provide your own zero-parameter constructor for classes you write.

L&C’s Maze Program

- I’m going to point out some mistakes in the Maze program from Chapter 4 of L&C, but first I should cop to my own mistake. I told you to use that program as a model for Project #2, without reading their code carefully. I assumed that they were solving the problem the way I wanted, but they were not, and their code provided a less helpful model for you. This made Project #2 harder than I intended.

- Their program looks for a path from the upper-left to the lower-right square in the maze, where the openness and seenness of each square is given by an int value in a two-dimensional array. They push any open and unseen neighbors of cells they visit onto a stack (of Position objects) and pop positions off in order to visit them.

- The good news is that if there is a path to the lower-right corner, they will not give up before finding it, and will return `true` from their `traverse` method.

More on L&C’s Maze

- The bad news is what happens if the program is called on a maze that has no path from top left to bottom right. Their `traverse` method has no way to exit its while loop in this case, except to pop all elements off of the stack and throw an `EmptyCollectionException`! I have to conclude that they never tested their code in this case.

- Another difference is that they do not maintain a valid path on the stack, as we did in Project #2. (They throw all open and unseen neighbors onto the stack at once, rather than one at a time.) Thus they can only tell whether a path exists, not what it is.

- Finally, there are at least two variables in their code that are defined but never used, suggesting that they somehow mixed up two versions of the program.

Java’s Stack Class -- The API

- Your best source of information about a class or interface already defined in Java is the **application programming interface** or **API**. The official Java API from Oracle (which acquired Java when it took over Sun Microsystems) is at http://download.oracle.com/javase/6/docs/api.

- This web site has an entry for every class and interface in Java, with its inheritance situation, fields, constructors, and methods.

- In the case of `Stack`, the API tells us that it extends `Vector`, which extends `AbstractList`, which extends `AbstractCollection`, which extends `Object`. Everything between `Vector` and `Object` is an abstract class, which cannot be instantiated.

- `Stack` has five methods of its own (`empty`, `peek`, `pop`, `push`, and `search)`, 43 inherited from `Vector` including `size`, and 13 inherited from other ancestors.

More on Java’s Stack Class

- Of course this allows a Stack object to do lots of things that an abstract Stack (something modeled by a StackADT variable in L&C’s set of definitions) cannot do.

- We can click on any method on the Stack page and find a description of it on the page for the class in which it is defined.

- If we want the actual code for these methods, we can download it from Oracle as long as we agree to the terms of the Java Research License. These allow us to use the code for research or educational purposes, but not to distribute it further to anyone who has not agreed to the JRL.

- There’s things we can learn from looking at the source code.

Java’s Stack Class -- The Source Code

- The code for `Stack` is actually pretty short because it essentially uses `Vector` to do the array implementation. The `Stack` methods are mostly one-line calls to more powerful `Vector` methods.

- A `Vector` object is pretty much like an `ArrayList`, a dynamically sized array. (All these classes are generic, of course.) One difference is that a `Vector` object can be downsized by the `trimToSize` method as well as upsized.

- The code for the `Vector` methods mostly works directly on the array in which the elements are stored, rather than by calling other methods.