CMPSCI 187: Programming With Data Structures

Lecture #10: Stacks in the Abstract
David Mix Barrington
26 September 2012

Stacks in the Abstract Exceptions

- What is a Stack?
- Collections of Objects
- Generic Classes
- Exceptions
- Programming by Contract
- The Two Stack Interfaces
- The Java `Stack` Class

What is a Stack?

- A **stack** is a data structure that holds a collection of objects. New objects can always be inserted with the **push** operation, but the only way to remove objects is with the **pop** operation, which removes and returns the *last object that was pushed*. We can look at the **top** object without removing it.

- Popping from an empty stack causes an error condition called **stack underflow**. We can avoid this by checking whether the stack is empty before every pop. Some stacks are **bounded**, meaning that they have a fixed **capacity**. Pushing onto a full stack causes an error condition called **stack overflow**, and we can detect a full stack before risking a push.

- Stacks were originally named after the receptacles for plates in a cafeteria, where only the top plate can be seen. They are widely useful in computing, probably most because they can be used to model **method calls**.

Examples of Using Stacks

- As we've discussed, when one method in Java calls another, control returns to the first method directly after the call, once the second method is finished. The operating system must remember the complete context of the first method so that it can restore that context when control returns. If the second method calls a third, the context of the second must also be stored. It is sensible to put the second context "above" the first on a stack, since we will not want to look at the first until the second has been restored. This process is particularly important for **recursive algorithms**, when the stack is storing multiple calls to the same method.

- If an ordered list items are all pushed onto a stack and then all popped off, they come off in **reverse order**. In Discussion #3 we used two stacks to **sort** a list of objects.

- Every left parenthesis in a mathematical text must be **balanced** by a right parenthesis, and different kinds of parentheses can nest but not interlock. We'll use a stack to check this balance, and then proceed to more complicated **parsing algorithms**.

Collections of Objects

- Our StringLog and StringBag classes could contain only strings, and it would be easy to write a StringStack class whose objects would be stacks of strings. But then we might find ourselves writing IntStack or MoveStack classes every time we wanted to put a new type of object into a stack.

- What if we built ObjectStack objects that could hold any kind of object? The problem with this idea is that when we take an Object off of an ObjectStack, the compiler has no information about what kind of object it is. If it were a String, then before we could run any String methods on it we would have to **cast** it into a variable of type String, e.g., `String w = (String) os.pop( ).` If the object popped is not a String, we get a `ClassCastException` at runtime.

- Since Java 5.0, Java has had a mechanism to create entire families of classes at once, where the family has a **type variable** and each value of that variable gives us a new class.

Generic Classes

- Let's start with an example of a **generic class**. A StringLog's basic behaviors were mostly independent of the fact that it held Strings. Suppose we had said:

```
public class Log<T> {
   private T[ ] log;
   private int lastIndex = -1;
...
```

- Then this code would create classes `Log<String>`, `Log<Integer>`, and so on, which we can think of as substituting those classes for all occurrences of "T" in the code of `Log<T>`. A method like `insert` would take a `T` as a parameter. We'd need to use T's `.equals` method, because all objects have this method but not all have `equalsIgnoreCase`.

- There are complications with generics that we should be able to ignore in this course for the most part.

Exceptions

- We've identified two **error conditions** that can come from using stacks, and we're familiar with many others such as an array index being out of bounds.

- We can try to avoid these conditions happening, for example by **guarding**, but Java gives us the capacity to deal with them without simply **crashing**.

- When we identify such a condition, we can define a class of **exceptions**, objects that come into existence when **thrown** and which crash the program unless **caught**. Many standard Java methods throw exceptions, and your own code can create them with a **throws** clause.

- If you don't want an exception to crash the program, you must catch them by enclosing the code where they occur with a **try block** and writing a *matching* **catch block** with the code to be run if the exception occurs.

Programming By Contract

- Writing code to deal with bad user input or other unusual conditions is a nuisance. It is easier to write code that assumes that certain **preconditions** are true when it is called.

- **Programming by contract** is the practice of writing methods that don't deal with the cases where their preconditions are true. If it's necessary to test the precondition of some, then it shouldn't be a precondition -- in this case we say that part of this method's job is to test for a particular **error condition**.

- The most common error condition is incorrect input given by some entity outside our program. Usually, then, we isolate interactions with such entities in particular methods. If I have a method `getInt` that needs to get an `int` from the console, the code that tells the user "Not an int, try again" should be within this method. Any other method should be able to assume that `getInt` will return an `int`. If the user doesn't give us one, and we wait forever for it, this amounts to a **denial of service attack**.

The Two Stack Interfaces

- DJW define three interfaces in order to have both bounded and unbounded stacks. BSI<T> and USI<T> differ in that only BSI<T> has an isFull method, and they have different **throws clauses**. (Since these exceptions are not **checked**, these throws clauses are actually just advice for the programmer.) Note that DJW's "pop" does *not* return the element popped -- you have to get it with "top" first if you want to save it.

```
public interface StackInterface<T> {
   void pop( ) throws StackUnderflowException;
   T top( ) throws StackUnderflowException;
   boolean isEmpty;}

public interface BoundedStackInterface<T> extends SI<T> {
   void push(T element) throws StackOverflowException;
   boolean isFull( );}

public interface UnboundedStackInterface<T> extends SI<T> {
   void push(T element);
```

The Java `Stack` Class

- The Java class library has a variety of data structures in `java.util`, called the **Collections package**. It's possible to teach CMPSCI 187 having you use these classes and interfaces exclusively, but we want you to get your hands dirty with the implementations of each structure as well as use them. Also, since it is commercial software, the Collections code is not always written so as to be easy for a student (or anyone else) to understand.

- In Collections, `Stack<T>` is a generic class, with methods `push`, `pop` (which is DJW's top and pop), `peek` (DJW's top), and `empty` (DJW's isEmpty). But it also has many other methods, some inherited from another class called `Vector<T>`. You can use `java.util.Stack<T>` in the same place you would use DJW's various Stack<T> classes, and if you never use the extra methods there is no particular problem. There is also the potential problem of the different names given to the standard stack operations.