Contents

0 Introduction

Recall that a data structure is a container object that stores related objects under a single name. These structures differ in the way they access, manage, acquire, and present data. We have seen them before. Let's round up a few examples from the Python and Java worlds.

Python has these key data structures.

1. `list` This data structure is a mutable heterogeneous sequence type. It holds a sequence of objects of any type. It uses the [] operator for access to items and to sublists.
2. `tuple` This data structure is an immutable version of the `list` data structure.
3. `dict` This data structure creates an associative table of key-value pairs. The keys and values must be hashable objects in Python.
4. `set` This is an unordered data structure that keeps a collection of objects without duplicates.

So far, we have seen these in Java.

1. `ArrayList<E>` This data structure is a mutable homogeneous sequence type. It holds a sequence of objects type `E`. It uses the get() method for access to items.
2. `arrays` These data structures are fixed-sized homogeneous mutable containers. Access to entries is given by the [] operator. `list` data structure.

We are now going to focus on data structures in Java. We will learn about the Java Collections Framework. This library contains powerful tools that will help you to accomplish programming tasks efficiently and easily.

But first, we will learn about data structures from the inside. To do this, we will need a fuller understanding of how generics and type parameters work. We will then apply this understanding to creating two implementations of a generic class for a stack. You will then extend these classes to a list implementation.

1 A Roundup of Basic Facts about Generics

We have been be using standard library classes for various data structures, which house collections of objects that are called *elements*. Each data structure has rules for item access and for adding items to the collection.

Recall that if we wanted an array list of strings we declared as follows.

```
ArrayList<String> roster = new ArrayList<>();
```

Informally we would say that the object pointed at by `roster` is an "`ArrayList` of strings." The contents of `<..>` constitute the type parameter of the `ArrayList`. We can use any object type as a type parameter for an `ArrayList`. We can also use the wrapper types if we wish to have an array list of a primitive type.

The type `ArrayList` without a type parameter is called a *raw type*. When writing new code, you should not use raw types. We shall see, however, that raw types can be used.

Let us proceed with an example. Consider this little program.

```
import java.util.ArrayList;

public class Blank
{
  public static void main(String [] args)
  {
    ArrayList foo = new ArrayList();
    foo.add("goo");
  }
}
```

Compilation of this code results in this compiler warning.

```
$ javac Blank.java
Note: Blank.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Let us now pry inside of this compiler jeremiad with the `-Xlint` option to get more detail.

```
$ javac -Xlint Blank.java
Blank.java:7: warning: [rawtypes] found raw type: ArrayList
    ArrayList foo = new ArrayList();
    ^
  missing type arguments for generic class ArrayList<E>
  where E is a type-variable:
    E extends Object declared in class ArrayList
Blank.java:7: warning: [rawtypes] found raw type: ArrayList
    ArrayList foo = new ArrayList();
                        ^
  missing type arguments for generic class ArrayList<E>
  where E is a type-variable:
    E extends Object declared in class ArrayList
Blank.java:8: warning: [unchecked] unchecked call to add(E) as a member of the raw type Arra
    foo.add("goo");
           ^
  where E is a type-variable:
    E extends Object declared in class ArrayList
3 warnings
```

We are getting an angry compiler response for using raw types. Clearly, this is not a current best practice. Java wants to to specify a type parameter for your array list, hence this warning.

Raw types work to ensure that Java is backwards-compatible. Prior to Java 5, you never specified a type parameter; you only worked with raw types. Objects stored in an `ArrayList` were... just `Objects`.

Now watch what happens when we try to gain access to the contents of our `ArrayList`.

```java
import java.util.ArrayList;

public class Blank
{
  public static void main(String [] args)
  {
    ArrayList foo = new ArrayList();
    foo.add("goo");
    System.out.printf("The length of %s is %d\n", foo.get(0),
        foo.get(0).length());

  }
}
```

We get a warning for using a raw type, and then we also get some irate scoldings from the compiler.

```
$ javac Blank.java
Blank.java:10: error: cannot find symbol
        foo.get(0).length());
                  ^
  symbol:   method length()
  location: class Object
Note: Blank.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
1 error
```

Now you might ask why we are getting an error. The call `foo.get()` returns an `Object`, not a `String`. Therefore, by the Visibility Principle, `foo.get().length()` makes no sense; you cannot compute the length of an `Object`. You can do this for a `String`. If you fetch an object from a container of raw type, you will just get an `Object`.

What is interesting is that the call to print `foo.get(0)` will compile and run (with the warning). Try it! This is because the `Object` class has a `toString()` method. The `Object` type object `foo.get()` points an a `String` with value `"goo"`. The variable sends the message to the `String` object and it prints itself.

The `Object` prints correctly because of the Delegation Principle. The type of a variable determines what methods are visible. The execution of those methods is delegated to the object. Since `"goo"` is a `String` object, it shows as a string and not as `Object@hexCrud`.

by creating a simple data structure ourselves, the Stack. This experience will help you understand what is happening when the Java Collections Framework is at work.

We will also put out stack to work by using it it to build an HP-style Reverse Polish Logic Calculator.

1.1 Type Erasure

Prior to Java 5, storage in collections such as the `ArrayList` worked a little like `ObjectOutputStreams`. The programmer could add object of any `Object` type to an `ArrayList`. Then, when fetching those values using the `get` method, the results had to be cast to the correct type to use them in their original form. In this system, the programmer is responsible for performing the correct casts.

We see this phenomenon at work in this `jshell` session.

```
jshell> ArrayList foo = new ArrayList();
foo ==> []

jshell> foo.add("syzygy");
|  Warning:
|  unchecked call to add(E) as a member of the raw type java.util.ArrayList
|  foo.add("syzygy");
|  ^---------------^
$2 ==> true

jshell> foo.add("muffins");
|  Warning:
|  unchecked call to add(E) as a member of the raw type java.util.ArrayList
|  foo.add("muffins");
|  ^----------------^
$3 ==> true

jshell> foo.get(0).length();
|  Error:
|  cannot find symbol
|    symbol:   method length()
|  foo.get(0).length();
|  ^---------------^

jshell> System.out.println(foo.get(0))
syzygy
```

```
jshell> System.out.println(foo.get(1))
muffins
```

The cast is syntactically clumsy and it is aesthetically execrable. And it was the old way of doing business.

In modern parlance, you do this instead.

```
jshell> ArrayList<String> foo = new ArrayList<>();
foo ==> []

jshell> foo.add("syzygy");
$2 ==> true

jshell> foo.add("muffins");
$3 ==> true

jshell> foo.get(0).length();
$4 ==> 6

jshell> foo.get(1).length();
$5 ==> 7
```

That ugly cast is now gone. Where'd it go? This is no mere detail. It is important you understand what is happening behind the scenes, so you fully understand the benefits, quirks and possible dangers entailed with using generics.

Again, you should always use generics when writing new code. However, it is important to understand what is happening, because you will see legacy code that does not use generics, and you need to know how to read and manage it correctly.

So how does that all work? Suppose you use an `ArrayList` with a type parameter. You compile the program. When you do, any calls that add items to the `ArrayList` are checked to see if they are adding items of the correct type. The compiler enforces this contract.

At run time all of these have the same appearance

- `ArrayList<Integer>`
- `ArrayList<ArrayList<Integer>>`
- the raw type `ArrayList`

Any calls that get items from the `ArrayList` have the appropriate casts added to them *by the compiler.* You are given the *Cast Iron Guarantee*, which says that all of the necessary casts will be added by the compiler.

Danger! You can void the Cast Iron Guarantee: The existence of an "unchecked warning" voids the guarantee.

If you are using generic classes, you cannot have an "unchecked warning;" this undermines the safety of your entire program. It is the responsibility of the creator of a generic class to extirpate any of these that might crop up.

When your code is in its run-time form, all evidence of generics is gone. All this code sees is the casts created by the compiler. Your code at runtime looks like pre-5 Java code, with raw types. This approach has two important benefits.

Pre-5 Java code will still compile. Java maintains backwards compatibility.

This erasure mechanism helps keep your byte code small. All Java `ArrayList`s look exactly the same at run time, so there is only one segment of code for `ArrayList`.

In contrast, C++ has a similar-looking mechanism called *templates*. A close analog to a Java `ArrayList` is C++'s `vector` class. If you make vectors with several different type parameters, object code will be generated for each one. This phenomenon is sometimes known as "code bloat." This is the opposite approach to generic programming to the type erasure approach of Java.

Be warned that there will be some pitfalls for you when writing generic code, especially if you deal with arrays. We will encounter this problem in our upcoming creation of our array based stack class. Take a look at this little class. In it, we attempt to create an array of generic type.

```java
public class UglySurprise<T>
{
  T[] myArray;
  public UglySurprise(int n)
  {
    myArray = new T[n];
  }
}
```

Now compile and you see an error. Java does not permit the creation of arrays of generic type.

```
$ javac UglySurprise.java
UglySurprise.java:6: error: generic array creation
    myArray = new T[n];
              ^
1 error
```

The reason for this will be revealed when we discuss subtyping and generics.

2 Inheritance and Generics

We will use the term *type* to refer to a class or an interface. We say S is a *subtype* of T for any of these three situations.

- S and T are interfaces and S is a subinterface of T.
- S and T is a class implementing S.
- S is a subclass of T.

If S is a subtype of T we will say that T is a *supertype* of S. Let us begin by creating these classes and compiling.

```
public class General
{
}
```

```
public class Specific extends General
{
}
```

```
import java.util.ArrayList;

public class ArrayTest
{
  public static void main(String[] args)
  {
    Specific [] s = new Specific[10];
    General [] g = new General[10];
    ArrayList<Specific> as = new ArrayList<Specific>();
    ArrayList<General> ag = new ArrayList<General>();
    testArray(s);
    testArray(g);
  }
  public static void testArray(General[] g)
  {
  }
  public static void testArrayList(ArrayList<General> ag)
  {
  }
}
```

Your code will compiles. This is because array types are *covariant*, i.e., that if S is a subtype of T, then S[] is a subtype of T[]. This seems intuitive and it does not come as any kind of terrible surprise.

Let us formulate analogous code for `ArrayList`s.

```
    testArrayList(as);
    testArrayList(ag);
```

Now compile this. Here is the result.

```
$javac ArrayTest.java
ArrayTest.java:13: error: incompatible types: ArrayList<Specific>
cannot be converted to ArrayList<General>
    testArrayList(as);
                  ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose
to get full output
1 error
```

Let's accept the invitation to dig.

```
$ javac -Xdiags:verbose ArrayTest.java
ArrayTest.java:13: error: method testArrayList in class ArrayTest cannot be applied to give
    testArrayList(as);
    ^
  required: ArrayList<General>
  found: ArrayList<Specific>
  reason: argument mismatch; ArrayList<Specific> cannot be converted to ArrayList<General>
1 error
```

Why the difference? This is because generics are *invariant*; i.e., if `S` is a subtype of `T`, then `ArrayList<S>` is *not* a subtype of `T` unless `S` and `T` are in fact the same type.

In `java.util` there is an interface called `List`. The standard `ArrayList` and `LinkedList` classes in `java.util` implement `List` so it is possible for `List` variables to point at `ArrayList` or `LinkedList` objects.

Let us learn what subtype relationships occur here. Modify `ArrayTest.java` as follows, so it will compile cleanly.

```
import java.util.ArrayList;
import java.util.List;

public class ArrayTest
{
  public static void main(String[] args)
  {
    Specific [] s = new Specific[10];
    General [] g = new General[10];
    ArrayList<Specific> as = new ArrayList<Specific>();
    ArrayList<General> ag = new ArrayList<General>();


  }
  public static void testArray(General[] g)
  {
  }
  public static void testArrayList(ArrayList<General> ag)
  {
  }
}
```

We now perform an experiment by adding this code to the `main` method.

```
List<General> lg = new ArrayList<General>();
```

This does compile. If `Foo` is a subtype of `Goo` and these two classes are generic, then `Foo<T>` is a subtype of `Goo<T>`. Notice that this breaks things.

```
List<General> lgs = new ArrayList<Specific>();
```

We show the resulting compiler error.

```
1 error found:
File: /Users/morrison/book/javaCode/j10/ArrayTest.java  [line: 13]
Error: /Users/morrison/book/javaCode/j10/ArrayTest.java:13:
incompatible types
found   : java.util.ArrayList<Specific>
required: java.util.List<General>
```

Notice that a `ArrayList<General>` is a subtype of `List<General>`, but things break because of invariance. An `ArrayList<Specific>` is not a subtype of `ArrayList<General>`

Programm Analysis Exercise

1. Create and compile this.

```
public class Suppress<T>
{
    Object[] myArray;
    int size;
    public Suppress(int size)
```

```
        {
          this.size = size;
          myArray = new Object[size];
        }
        public T get(int k)
        {
            if( k < 0 || k >= size)
            {
                throw new IndexOutOfBoundsException();
            }
            return (T) myArray[k];
        }
        public void set(int k, T newObject)
        {
            myArray[k] = newObject;
        }
    }
```

2. Open this in `jshell` and execute these commands. What sort of compiiler grumblings do you see?

   ```
   Suppress<String> s = new Suppress<>(10);
   s.put(0, "foo");
   s.set(0, "foo");
   s.get(0);
   s.get(0).getClass()
   ```

3. Can you argue this case? *It is guaranteed that only items of class `T` can be added to our data structure here?* Try and use `set` on the wrong type of object. What happens?

4. There is a java annotation `@SuppressWarnings("unchecked")`. Use it to silence the compiler grumblings by applying it to the approriate methods.

3 What is a Stack?

A stack is a first-in-first-out (FIFO) data structure. The rule is that you interact with just the top of the stack; items underneath should be invisible to the client programmer. This concept has appeared to us in a couple of different guises.

This idea should be familiar from your early encounters with functions. When a function in Python or Java is called, its *stack frame* that contains its parameters and local variables is placed on the top of the call stack. When the function returns, its frame is removed from the top of the call stack and it is destroyed. Only the function on the top of this stack is visible. Programs in Java, C and Python all have “main routines”. The program ends its execution when the main routine is removed from the call stack.

Here are two examples where a stack could come in very handy. Imagine you have an expression such as

$$(((1+x^2)^3)+\sin(x)(x-(x-5)^3)).$$

Strip out the parens to see

$$((())()(())).$$

Let us see how we can use a stack to see if the parentheses in here constitute a valid scheme of association. Do the following

1. Make an empty stack.
2. Place an open parenthesis onto the stack.
3. When you encounter a closed parenthesis, look at the stack. If it is not empty, remove the open parenthesis from the top of the stack. If the stack is empty, you have a closed parenthesis with no corresponding open parenthesis; conclude that the expression is malformed.
4. When you have processed all of the parentheses, the stack should be empty. If it is, your expression has a clean bill of health. If there are leftovers on the stack, you have unclosed parentheses, and the expression is malformed.

Here is another example. You can check if an HTML document is well-formed using a stack of strings. Do the following.

1. Make an empty stack.
2. When a tag opens, obtain its type and put it on the stack.
3. When you encounter a closed tag, look at the stack. If the tag on the top does not match the closing tag, report an error. You have mismatched tags. If it does match, remove the top tag from the stack. If the stack is empty, you have an unmatched closing tag.
4. When you have processed all of the tags, the stack should be empty. If it is, your document is well-formed If there are leftovers on the stack, you have unclosed tags, and the document is malformed.

We are going to create a stack via two means. In one, we will use an array and in the other we will create a *linked data structure*. Both of these will grow dynamically to accommodate a large number of elements without the user having do deal with any resizing operation.

4 A Stack Interface

We begin by creating `IStack`, an interface for a stack class. We will then code two implementations to this interface, one based on an array and the other will be based on links.

Note that we will be creating a generic interface; the type parameter `E` denotes the types of entries (data) we expect in the stack.

Our stack class will have the following features.

- `pop` This method will return the top item on the stack and then remove it. An exception will be thrown for popping from an empty stack.
- `peek` This method will return the top item on the stack, and leave the stack unchanged. Stack objects throw an exception if the client attempts to peek at an empty stack.
- `push` This method will place the specified item onto the stack.
- `isEmpty` This method will return `true` when the stack is empty. A client programmer should use this if there is danger of peeking at or popping from an empty stack.
- `clear` This method will discard all elements residing on the stack.

The names `pop`, `push` and `peek` are the universally standard names for these operations. We now specify the method heads we need formally. Our class will be generic with type parameter `E`. So `pop` will have the following head.

```
public E pop();
```

Now we proceed with `push`. It will place an item of type `E` onto the stack. So its method header will be

```
public void push(E newItem);
```

The `peek` method will just look at the top item on the stack and return it. Hence its header has the following appearance.

```
public E peek();
```

The `clear` method is simple; it takes no arguments, returns nothing and just removes all elements from the stack. Its header is

```
public void clear();
```

Finally we check for emptiness; no argument is required and a `boolean` is returned.

```
public boolean isEmpty();
```

Assembling this, we have the following interface.

```
public interface IStack<E>
{
    public E pop();
    public void push(E newItem);
    public E peek();
    public void clear();
    public boolean isEmpty();
}
```

To cement our design decisions, we now javadoc this interface; both the link based and the array based versions of the stack will implement this common interface.

```
public interface IStack<E>
{
    /**
    *   This removes the top element on the stack and returns it.
    *   @return top element
    *   @throws EmptyStackException if the user attempts to pop from
    *   an empty stack.
    */
    public E pop();
    /**
    *   This removes the top element on the stack and returns it.
    *   @param newElement the new element we are pushing onto this Stack.
    */
    public void push(E newItem);
    /**
    *   This returns the top element from this stack..
    *   @return top element
    *   @throws EmptyStackException if the user attempts to peek at
    *   an empty stack.
    */
    public E peek();
    /**
    *   This clears the stack of elements.
    */
    public void clear();
    /**
    *  This returns <code>true</code> if the stack is empty.
    *  @return true if the stack is empty.
    */
    public boolean isEmpty();
}
```

5 Creating A Link-Based Stack Class

Our first version of a stack class will be based upon a objects called *links*; each link will contain a datum and a pointer to the next link. Our stack will have a bottom link (which will point at `null`). Each link will point down at the next lower item on the stack.

We begin by creating a shell for the class. Notice the use of the type parameter, since we are creating a generic class. We will name this `LStack`, since it is will be a link-based stack.

```
public class LStack<E>
{
}
```

Let us begin by stubbing in the methods specified by `IStack`. You should immediately compile this. Place it in a directory with `IStack` and compile.

```
public class LStack<E> implements IStack<E>
{
    public E pop()
    {
        return null;
    }
    public void push(E newItem)
    {
    }
    public E peek()
    {
        return null;
    }
    public void clear()
    {
    }
    public boolean isEmpty()
    {
        return false;
    }
}
```

Notice that we have made no specification of any kind for the implementation. This shell will work for both `LStack`, the link-based stack and `AStack`, the array-based implementation that we will implement later.

6 Building the Link Class

We begin by specifying state. We said the link will contain a datum and a means for finding the next link. To do this, we provide the following state. This class will be a non-public class that lives in the same file as `LStack`. So, create a file named `LStack.java` and place this code in it.

```java
public class LStack<E> implements IStack<E>
{
    public E pop()
    {
        return null;
    }
    public void push(E newItem)
    {
    }
    public E peek()
    {
        return null;
    }
    public void clear()
    {
    }
    public boolean isEmpty()
    {
        return false;
    }
}
class Link<E>
{
    //code for the link class
}
```

Now compile. We place the `Link` class in the same file as `LStack` since it is a class that exists to serve `LStack`. We do not make the `Link` class an inner class, since it needs no access to the enclosing class's state variables.

We now go to work on the link class. We produce two constructors for the link class to initialize the state variables.

```java
class Link<E>
{
    private E datum;
    private Link<E> next;
    public Link(E datum, Link<E> next)
    {
        this.datum = datum;
        this.next = next;
    }
    public Link(E datum)
    {
        this(datum, null);
    }
}
```

Finally, we make getters obtain the state of a link. We can add setters later if we see the need for them.

```
class Link<E>
{
    private E datum;
    private Link<E> next;
    public Link(E datum, Link<E> next)
    {
        this.datum = datum;
        this.next = next;
    }
    public Link(E datum)
    {
        this(datum, null);
    }
    public E getDatum()
    {
        return datum;
    }
    public Link<E> getNext()
    {
        return next;
    }
}
```

7 Deciding on Methods for `LStack`

We will now determine what methods our stack data structure will have and how they will work. Both stack data structures we create will implement exactly this interface. Note that we will create an exception class for illegal interactions with an empty stack. This is simple to create. Just create a file named `EmptyStackException.java` and place this code in it.

```
public class EmptyStackException extends RuntimeException
{
}
```

We had decided on these methods.

- `boolean isEmpty()` This evaluates to `true` if the stack has no elements in it.
- `E peek()` This returns the top item on the stack. Notice, it returns the *datum*, not the link containing it. We throw an `EmptyStackException` if the client attempt to peek on an empty stack.
- `E pop()` This returns the top item on the stack and remove it. This entails getting rid of the link at the top. We throw an `EmptyStackException` if the client attempt to pop from an empty stack.
- `void push(T newItem)` This wraps the new item in a link and places it on the top of the stack.

We will also put in a `toString()` method so we can see what we are doing as we code. This method will print the stack from top to bottom.

So our complete file is as follows. So far we have just stubbed in the methods so that the class will compile.

```
public class LStack<E> implements IStack<E>
{
    public E pop()
    {
        return null;
    }
    public void push(E newItem)
    {
    }
    public E peek()
    {
        return null;
    }
    public void clear()
    {
    }
    public boolean isEmpty()
    {
        return false;
    }
    @Override
    public String toString()
    {
        return "";
    }
}
class Link<E>
{
    private E datum;
    private Link<E> next;
    public Link(E datum, Link<E> next)
    {
        this.datum = datum;
        this.next = next;
    }
    public Link(E datum)
    {
        this(datum, null);
    }
    public E getDatum()
    {
        return datum;
    }
    public Link<E> getNext()
    {
        return next;
    }
}
```

8 Implementing the Link-Based Stack, `LStack`

Now let us turn our attention to our skeleton stack class. It needs to know about the top of the stack, so we insert the state variable

```
    Link<E> top;
```

to point at the top of the stack. Let us agree that when the stack is empty, we will have `top` set to `null`. We also give our class an appropriate constructor.

```
public LStack()
{
    top = null;
}
```

Our stacks will be born empty.

We now know when a stack is empty; this is so when `top == null`. We now implement the method `isEmpty()`

```
public boolean isEmpty()
{
    return top == null;
}
```

Next, we will work on the `push` method. This method takes an object `newItem` of type `E`. Internally, our stack does not accept bare items; these items must be clothed in a link. So we begin by doing that.

```
    public void push(E newItem)
    {
        top = new Link<E>(newItem, top);
    }
```

This is a complex arabesque; we shall step through what happens here. We create a new link that holds our new item and which points at the existing top of the stack; if the stack is empty, this us just `null`. Otherwise, the new item is interposed at the top of the stack and the item at the top points at the old stack. You should draw a picture and see how this works.

What is needed next is for us to be able to see that stuff is being pushed onto the stack. The need to implement `toString` arises. Here is the idea. Start at the top. If it is null, we do nothing; we will just show an empty list like so: `[]`. Otherwise we get the datum, print it, and go to the next link. We stop when we encounter a `null` link. Notice that we make a copy of `top` to iterate with; we do not want `top` itself iterating to the bottom and losing our stack. The `toString()` method, by convention, is an accessor method and should not change the state of the stack.

```
    @Override
    public String toString()
    {
        if(isEmpty())
            return "[]";
        StringBuffer sb = new StringBuffer();
        Link<E> foo = top;
        sb.append("[");
        while(foo.getNext() != null)
        {
            sb.append(foo.getDatum());
            foo = foo.getNext();
        }
        sb.append(foo.getDatum() + "]");
        return sb.toString();
    }
```

Let us now create a driver class to test our `LStack` class. Create this program, `Driver.java`.

```
public class Driver
{
    public static void main(String [] args)
    {
        System.out.println("LStack Class Driver");
        LStack<String> s = new LStack<String>();
        s.push("a");
        s.push("b");
        s.push("c");
        s.push("d");
        System.out.println(s);
    }
}
```

You should see this result.

```
[d, c, b, a]
```

The items are “backwards” because a stack is a LIFO (last in first out) data structure. The top of the stack is the leftmost entry displayed.

Now let us peek. If a stack is empty, and you peek, this is likely a programmer goof. If a hapless client attempts to peek at an empty stack, we shall hand him an `EmptyStackException` for his troubles. Next time he will have the manners to use `isEmpty()` for its intended purpose. This exception we throw will is a runtime exception.

We now begin our `peek` method.

```
    public E peek() throws EmptyStackException
    {
        if(isEmpty())
        {
            throw new EmptyStackException();
        }
        return null;
    }
```

Peeking is easy. If the stack is not empty, you just get the top’s datum. We insert this in the `return` statement.

```
    public E peek() throws EmptyStackException
    {
        if(isEmpty())
        {
            throw new EmptyStackException();
        }
        return top.getDatum();
    }
```

Once we get `peek`, it is straightforward to create `pop`. We will throw an exception if we see an attempt to pop from an empty stack. Then we will save the top element, remove it, and then finally return it.

```
    public E pop() throws EmptyStackException
    {
        if(isEmpty())
        {
            throw new EmptyStackException();
        }
        E out = top.getDatum();  //fetch top item
        top = top.getNext();     //amputate
        return out;              //return top item
    }
```

Now let us add a little torture test in a driver class to see if this works.

```
public class Driver
{
    public static void main(String[] args)
    {
        LStack<String> s = new LStack<String>();
        for(int k = 0; k < 10; k++)
        {
            s.push("" + k);
        }
        s.seymour();
        for(int k = 0; k < 10; k++)
        {
            s.pop();
        }
        if(s.isEmpty())
            System.out.println("PASS");
    }
}
```

Now run. Here we see the action occurring in a terminal window.

```
$ java Driver
9
8
7
6
5
4
3
2
1
0
PASS
```

Et Voila! Now try this with 10,000 elements and watch it work.

9 Iterator and Iterable in the Link-Based Stack

Wouldn't it be cool if you could use a collections `for` loop on a stack? It would be helpful if we could create our own data structure and have a `for` loop walk through them cleanly. This feature exists for the standard `ArrayList`. Why not for our custom structures?

To accomplish this requires two simple steps. Your class must implement the interface `Iterable<E>`. We look in the API guide and we see that one method is required.

```
    public Iterator<E> iterator();
```

Now we ask, "What is an iterator?" We look in the API guide and see that, it too, is an interface. This interface has three methods.

```
public Iterator<E>
{
    public E next();
    public boolean hasNext();
    public E remove();
}
```

It is an accepted standard that `remove()` is optional. We can render it unappetizing by throwing an `UnsupportedOperationException()`. That is the tack we shall take here.

To get our stack class ready we modify its header to read

```
public class LStack<E> implements IStack<E>, Iterable<E>
```

We can keep it compiling by adding the iterator method as follows.

```
    public iterator()
    {
        return null;
    }
```

We will implement our iterator as an inner class. We begin by making a hollow class with the required methods. While we are here, let us also administer the horse-kick if the hapless client calls `remove()`.

```
    class Navigator implements Iterator<E>
    {
        private Link<E> nav;
        public Navigator()
        {
        }
        public void remove()
        {
            throw new UnsupportedOperationException();
        }
        public boolean hasNext()
        {
        }
        public E next()
        {
        }
    }
```

You will need to add the import statement

```
import java.util.Iterator;
```

Now place this inner class inside of your Stack class and it will compile. The job of `nav` is to navigate through our stack. It is easy to write `hasNext()`

```
        public boolean hasNext()
        {
            return nav != null;
        }
```

When writing `next()`, we must return the datum then move to the next node. It goes like this.

```
        public E next()
        {
            E out = nav.getDatum();
            nav = nav.getNext();
            return out;
        }
```

You now have your iterator. Now return to the `iterator` method for the enclosing `Stack` class and you have

```
    public Iterator<E> iterator()
    {
        return new Navigator();
    }
```

We now show the entire program.

```
import java.util.Iterator;
public class LStack<E> implements IStack<E>, Iterable<E>
{
    Link<E> top;
    public LStack()
    {
        top = null;
    }
    public void push(E newItem)
    {
        Link<E> newLink = new Link<E>(newItem);
        if(isEmpty())
        {
            top = newLink;
        }
        else
        {
            newLink.setNext(top);
            top = newLink;
        }
    }
    public E pop() throws EmptyStackException
    {
        if(isEmpty())
        {
            throw new EmptyStackException();
        }
        E out = top.getDatum();//fetch top item
        top = top.getNext();  //amputate
        return out;//return top item
    }
```

```
public E peek() throws EmptyStackException
{
    if(isEmpty())
    {
        throw new EmptyStackException();
    }
    return top.getDatum();
}

public boolean isEmpty()
{
    return top == null;
}
@Override
public void toString()
{
    Link<E> foo = top;
    while(foo != null)
    {
        System.out.println(foo.getDatum());
        foo = foo.getNext();
    }
}

public Iterator<E> iterator()
{
    return new Navigator();
}
/*************************Inner Iterator Class**********************/
class Navigator implements Iterator<E>
{
    Link<E> nav;
    public Navigator()
    {
        nav = top;
    }
    public void remove()
    {
        throw new UnsupportedOperationException();
    }
    public boolean hasNext()
    {
        return nav != null;
    }
    public E next()
    {
            E out = nav.getDatum();
            nav = nav.getNext();
            return out;
        }
    }
}
```

We now have a full-featured link-based stack class. Now let's test the for loop

```
public class Driver
{
    public static void main(String[] args)
    {
        Stack<String> s = new Stack<String>();

        for(int k = 0; k < 10; k++)
        {
            s.push("" + k);
        }
        System.out.println("collections for loop");
        for(String q: s)
        {
            System.out.println(q);
        }
        System.out.println("Here are the stack elements.");
        s.seymour();
        for(int k = 0; k < 10; k++)
        {
            s.pop();
        }
        if(s.isEmpty())
            System.out.println("PASS");
    }
}
```

Running this we see

```
$ java Driver
collections for loop
9
8
7
6
5
4
3
```

```
2
1
0
Here are the stack elements.
9
8
7
6
5
4
3
2
1
0
PASS
$
```

10 Programming Project: A Linked List Class, `LList.java`

In this project, you will write a simplified version of Java's `LinkedList` class. The purpose of this project is to acquaint you further with link-based structures and for you to get your hands dirty with them.

Here is the interface.

We begin by creating the interface `IList<E>` and have your `LList<E>` implement this interface.

```java
import java.util.Iterator;
public interface IList<T>
{
    /**
    *  appends <code>item</code> to this list
    *  @param item object to be added to this list
    *  @return true if <code>item</code> is added to this list
    */
    public boolean add(T item);
    /**
    *  appends <code>item</code> to this list
    *  @param index The object <code>item</code> is inserted
    *  at <code>index</code>.
    *  @param item object to be added to this list
    *  @return true if <code>item</code> is added to this list
    */
```

```
public boolean add(int index, T item);
/**
*  This returns a -1 if <code>quarry</code> is not found
*  or the index of the fist instance of <code>quarry</code>
*  otherwise.
*  @param quarry object to be added to this list
*  @return -1 if <code>quarry</code> is not found or the
*  index where the first instance of <code>quarry</code> is located.
*/
public int indexOf(T quarry);
/**
*  This removes the first instance of <code>quarry</code> from this
*  list; if no instance is found, it returns <code>false</code>.
*  @param quarry object to be removed to this list
*  @return <code>false</code> if <code>quarry</code> is not found
*  or the index where
*  the first instance of <code>quarry</code> is located.
*/
public boolean remove(T quarry);
/**
*  This removes the object found at index <code>index</code>
*  @param quarry object to be removed to this list
*  @return <code>false</code> if <code>quarry</code> is
*  not found  or the index where the first instance of
*  <code>quarry</code> is located.
*  @throws IndexOutOfBoundsException if an out-of-bounds
*  index is passed
*/
public T remove(int index);
/**
*  This removes all objects equal to <code>quarry</code> from this list.
*  @param quarry object to be extirpated from  this list
*  @throws IndexOutOfBoundsException if you attempt to remove a nonexistent
*  entry
*  @return <code>false</code> if <code>quarry</code> is not found
*/
public boolean extirpate(T quarry);
/**
*  This removes all elements from this list.
*/
public void clear();
/**
*  This computes the size of the list
*  @return the number of elements in this list
*/
public int size();
```

```
    /**
    *   This replaces the object at index <code>index</code>
    *   with <code>newItem</code>.
    *   @param index the index at which we are performing the replacement
    *   @param newItem the new item we are placing into the list
    *   @return the item being replaced [the evictee]
    *   @throws IndexOutOfBOundException if the <code>index</code>
    *   is out of bounds.
    */
    public void set(int index, T newItem);
    /**
    *   This fetches the object at index <code>index</code>.
    *   @param index the index from which we wish to retrieve an element.
    *   @return the element at <code>index</code> in this list.
    *   @throws IndexOutOfBOundException if the <code>index</code> is
    *   out of bounds.
    */
    public T get(int index);
    /**
    *   This checks for the presence of <code>quarry</code> in this list.
    *   @param quarry the item we are searching for in this list
    *   @return <code>true</code> if <code>quarry</code> is found in this list
    */
    public boolean contains(T quarry);
    /**
     * This creates an iterator that walks through this list in index order.
     * @return iterator for this list.
     */
    public Iterator<T> iterator();
}
```

This is a class implementing the interface `IList<T>` with the required methods stubbed in.

```
import java.util.Iterator;
public class LList<T> implements IList<T>, Iterable<T>
{
    Link<E> head;
    public LList()
    {
        head = null;
    }
    /**
    *  appends <code>item</code> to this list
    *  @param item object to be added to this list
    *  @return true if <code>item</code> is added to this list
```

```java
*/
public boolean add(T item)
{
    return false;
}
/**
*  appends <code>item</code> to this list
*  @param index The object <code>item</code> is inserted at <code>index</code>.
*  @param item object to be added to this list
*  @return true if <code>item</code> is added to this list
*/
public boolean add(int index, T item)
{
    return false;
}
/**
*  This returns a -1 if <code>quarry</code> is not found or the index of
*  the fist instance of <code>quarry</code> otherwise.
*  @param quarry object to be added to this list
*  @return -1 if <code>quarry</code> is not found or the index where
*  the first instance of <code>quarry</code> is located.
*/
public int indexOf(T quarry)
{
    return 0;
}
/**
*  This removes the first instance of <code>quarry</code> from this
*  list; if no instance is found, it returns <code>false</code>.
*  @param quarry object to be removed to this list
*  @return <code>false</code> if <code>quarry</code> is not found
*  or the index where
*  the first instance of <code>quarry</code> is located.
*/
public boolean remove(T quarry)
{
    return false;
}
/**
*  This removes the object found at index <code>index</code>
*  @param quarry object to be removed to this list
*  @return <code>false</code> if <code>quarry</code> is not found
*  or the index where
*  the first instance of <code>quarry</code> is located.
*  @throws IndexOutOfBoundsException if an out-of-bounds index is passed
*/
```

```
public T remove(int index)
{
    return null;
}
/**
*  This removes all objects equal to <code>quarry</code> from this list.
*  @param quarry object to be extirpated from  this list
*  @return <code>false</code> if <code>quarry</code> is not found
*/
public boolean extirpate(T quarry)
{
    return false;
}
/**
*  This removes all elements from this list.
*/
public void clear()
{
}
/**
*  This computes the size of the list
*  @return the number of elements in this list
*/
public int size()
{
    return 0;
}
/**
*   This replaces the object at index <code>index</code> with <code>newItem</code>.
*   @param index the index at which we are performing the replacement
*   @param newItem the new item we are placing into the list
*   @return the item being replaced [the evictee]
*   @throws IndexOutOfBOundException if the <code>index</code> is out of bounds.
*/
public void set(int index, T newItem)
{

}
/**
*   This fetches the object at index <code>index</code>.
*   @param index the index from which we wish to retrieve an element.
*   @return the element at <code>index</code> in this list.
*   @throws IndexOutOfBOundException if the <code>index</code> is out of bounds.
*/
public T get(int index)
{
        return null;
    }
    /**
    *   This checks for the presence of <code>quarry</code> in this list.
    *   @param quarry the item we are searching for in this list
    *   @return <code>true</code> if <code>quarry</code> is found in this list
    */
    public boolean contains(T quarry)
    {
        return false;
    }
    /**
     * This creates an iterator that walks through this list in index order.
     * @return iterator for this list.
     */
    public Iterator<T> iterator()
    {
        return null;
    }
}
class Link<E>
{
    private E datum;
    private Node<E> next;
    public Node(E datum, Node<E> next)
    {
        this.datum = datum;
        this.next = next;
    }
    public Link(E datum)
    {
        this(datum, null);
    }
    public E getDatum()
    {
        return datum;
    }
    public Link<E> getNext()
    {
        return next;
    }
}
```

Use the link-based stack as a guide to help you get started. Implement the methods in this shell class and test them to see that you have a fully functioning stack. Notice that the client knows nothing about the internal workings of the stack; in particular the client programmer has no idea you are using links. He simply pushes, pop, and writes his application using your class.

11 An Array Based Stack

Recall the interface we prescribed for a stack.

```
import java.util.Iterator;
public class AStack<E> implements Iterable<E>
{
    public AStack()
    {
    }
    public void push(E newItem)
    {
    }
    public E pop()
    {
        return null;
    }
    public E peek()
    {
        return null;
    }

    public boolean isEmpty()
    {
        return false;
    }
    public int size()
    {
        return -3;
    }
    public void seymour()
    {
    }
    public Iterator<E> iterator()
    {
        return null;
    }
}
```

We are going to create a stack class for this interface, but instead of using links, we will use an array to hold our elements. This presents a challenge. We must beware of overflowing the array and we must resize it when it gets nearly full. Along the way, we will have an encounter with the phenomenon of *type erasure*, which will render our challenge more difficult. Nevertheless, we shal circumvent this will a clever arabesque. Now strap on your seat belt.

Let us begin by creating two integer state variables. We will use `capacity` to refer to the size of the private array we shall create. We will create the private variable `size` to point just above the top of the stack. So, when the stack is empty, we make `size` 0. Now we have a some reasonable state variables, so we will create this and make the constructor.

```
    private int capacity;
    private int size;
    private E[] entries;
    public AStack(int _capacity)
    {
        capacity = _capacity;
        if(capacity < 1)
            throw new IllegalArgumentException();
        entries = new E[capacity];
        size = 0;
    }
    public AStack()
    {
        this(10);
    }
```

Notice we created two constructors. The default gives you an initial capacity of 10. This imitates the action of Java's array list type. The other allows you to specify a capacty of any positive integer size. We also burn the inept with an exception when they do something stupid, like make a zero or negative capacity.

Let us compile our masterpiece. We find ourselves on the business end of compiler ire.

```
$ javac AStack.java
AStack.java:13: error: generic array creation
        entries = new E[capacity];
                  ^
1 error
$
```

It is time for an arabesque. As it turns out, we cannot create arrays of generic type. We defer the discussion of the reason until we discuss type erasure. If you can't bear the suspense, flip ahead and take a peek. Otherwise calmy accept the upcoming turn of events.

Insted of an `E()`, let us us an array of `Object`s. So we modify our code as follows.

```
private int capacity;
private int size;
private Object[] entries;
public AStack(int _capacity)
{
    capacity = _capacity;
    if(capacity < 1)
        throw new IllegalArgumentException();
    entries = new Object[capacity];
    size = 0;
}
public AStack()
{
    this(10);
}
```

All traces of compiler dyspepsia now vanish. We will have to enforce the type restriction on items being added to this array ourselves, but you will see that the interface will do this for us nicely. We go for the easy pickings first. When is the stack empty? When its size is zero. What is its size? We know that. So here we implement those two methods with one-liners.

```
public boolean isEmpty()
{
    return size == 0;
}
public int size()
{
    return size;
}
```

Now let us write `peek`. If the stack is empty, we throw an exception. Otherwise, we just look at the top item in the stack. Remember, `size` points just aboe the stack, so note our use of `size - 1` to get it correct.

```
public E peek()
{
    if(isEmpty())
        throw new EmptyStackException();
    return (E) entries[size - 1];
}
```

Now compile. See the following squawling.

```
$ !javac
javac AStack.java
Note: AStack.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
$
```

We are doing something kind of edgy but we know what we are doing, so we will add this annotation

```
@SuppressWarnings("unchecked")
```

above the `peek` method. The compiler will no longer grumble. This grumbling was triggered by the cast to `E` in the return statement.

Our push method will check the type of items placed on the stack. It will be the only entry point for new items to go onto the stack. As a result, we have a cast-iron guarantee that only items of type `E` will ever appear on our stack. That is what makes it safe to use the `@SuppressWarnings("unchecked")` annotation.

Now we shall pop. This requires us to keep track of the item being popped, then we can lower the size by 1. We could leave the popped item in the array, but we will set it to null to encourage garbage collection.

Next we will push new items onto the stack. The push method is the gatekeeper. We shall insist all items we push be type `E`. This is checked at compile time by the `push` method. If you attempt to push an item of the wrong type on the stack, the comipiler will issue forth with error messages and disapprobation.

So heere is how we will approach it. If the client attempts to pop from an empty stack, an exception will be thrown. We then make a record of the top item on the stack. Next, we will set the entry on the stack referring to the top item to `null` to encourage garbage collection. Finally, we decrement size and return the item we recorded. Here is the implementation

```
@SuppressWarnings("unchecked")
public E pop()
{
    if(isEmpty())
        throw new EmptyStackException();
    E out = (E) entries[size - 1];
    entries[size - 1] = null; //encourage garbage collection
    size--;
    return out;
}
```

We need the annotation because of the cast. We are guaranteed, because of the contract we have for `push` that only items of type `E` will be placed on the stack.

Now let us implement `seymour`. That's easy.

```
    public void seymour()
    {
        int k;
        for(k = size - 1; k >= 0; k--)
        {
            System.out.println(k);
        }
    }
```

We now administer a torture test.

```
public class ADriver
{
    public static void main(String[] args)
    {

        AStack<String> s = new AStack<String>();
        for(int k = 0; k < 10; k++)
            s.push("" + k);
        s.seymour();
        for(int k = 0; k < 10; k++)
            s.pop();
        if(s.isEmpty())
            System.out.println("PASS");

    }
}
```

Here is the result.

```
$ java ADriver
9
8
7
6
5
4
3
2
1
0
PASS
```

Finally, we now write the iterator. It will be an inner class. We also modify the iterator method so it is not returning a null object. If you forget you will get a `NullPointerException` when you test the collections `for` loop.

```
    public Iterator<E> iterator()
    {
        return new Navigator();
    }
    /******************iterator class******************************/
    class Navigator implements Iterator<E>
    {
        int nav;
        public Navigator()
        {
            nav = size;
        }
        public void remove()
        {
            throw new UnsupportedOperationException();
        }
        public boolean hasNext()
        {
            return nav != 0;
        }
        @SuppressWarnings("unchecked")
        public E next()
        {
            nav--;
            return (E) entries[nav];
        }
    }
```

We now show the class in its entirety.

```
import java.util.Iterator;
public class AStack<E> implemnets Iterable<E>
{

    private int capacity;
    private int size;
    private Object[] entries;
    public AStack(int _capacity)
    {
        capacity = _capacity;
        if(capacity < 1)
            throw new IllegalArgumentException();
        entries = new Object[capacity];
        size = 0;
    }
    public AStack()
```

```
{
    this(10);
}
public void push(E newItem)
{
    entries[size] = newItem;
    size++;
}
@SuppressWarnings("unchecked")
public E pop()
{
    if(isEmpty())
        throw new EmptyStackException();
    E out = (E) entries[size - 1];
    entries[size - 1] = null; //encourage garbage collection
    size--;
    return out;
}
@SuppressWarnings("unchecked")
public E peek()
{
    if(isEmpty())
        throw new EmptyStackException();
    return (E) entries[size - 1];
}

public boolean isEmpty()
{
    return size == 0;
}
public int size()
{
    return size;
}
@Override
public String toString()
{
    if(isEmpty())
        return "[]";
    StringBuffer sb = new StringBuffer();
    int k;
    for(k = size - 1; k >= 1; k--)
    {
        sb.append(k + ", ");
    }
    sb.append(k + "]");
```

```
        return sb.toString();
    }
    public Iterator<E> iterator()
    {
        return Navigator;
    }
}
```

Modify the driver by removing the call to `toString()` and using the collections `for` loop as follows.

```
public class ADriver
{
    public static void main(String[] args)
    {

        AStack<String> s = new AStack<String>();
        for(int k = 0; k < 10; k++)
            s.push("" + k);
        for(String q: s)
        {
            System.out.println(q);
        }
        for(int k = 0; k < 10; k++)
            s.pop();
        if(s.isEmpty())
            System.out.println("PASS");
    }
}
```

This will work just as it did before we put the iterator in place.

12 Some Perspective

We have just implemented the same interface, that of a stack, in two very different ways. Each of these two methods has its strengths and weaknesses. In the next chapter we look at the performance of algorithms performed inside of data structures. By analyzing the performance characteristics of a particular implementatoin of an algorthm, we can understand when it is appropriate to choose a particular implementation of an interface.

Before we begin our analysis of algorithms in the next chapter we will explain some technical matters involved in generic programming. There are big benefits to this approach, but there are also hazards it is wise to be aware of.

13 Programming Project: A Array-Based List Class, `AList.java`

In this project, you will write a simplifed version of Java's `ArrayList` class. The purpose of this project is to acquaint you further with array-based structures and for you to get your hands dirty with them. You will implement the interface `IList<E>`.

This is a class implementing the interface `IList<T>` with the required methods stubbed in.

```java
import java.util.Iterator;
public class AList<T> implements IList<T>, Iterable<T>
{
    //here is where you store the array for the stack
    private Object[] entries;
    //This is the size of the stack's array.
    private int capacity;
    //this points at the top of the stack.
    private int size;
    public AList()
    {
        capacity = 10;  //ArrayList's default
        entries = new Object[capacity];
        size = 0;
    }
    /**
    *  appends <code>item</code> to this list
    *  @param item object to be added to this list
    *  @return true if <code>item</code> is added to this list
    */
    public boolean add(T item)
    {
        return false;
    }
    /**
    *  appends <code>item</code> to this list
    *  @param index The object <code>item</code> is inserted at <code>index</code>.
    *  @param item object to be added to this list
    *  @return true if <code>item</code> is added to this list
    */
    public boolean add(int index, T item)
    {
        return false;
    }
```

```
/**
*  This returns a -1 if <code>quarry</code> is not found or the index of
*  the fist instance of <code>quarry</code> otherwise.
*  @param quarry object to be added to this list
*  @return -1 if <code>quarry</code> is not found or the index where
*  the first instance of <code>quarry</code> is located.
*/
public int indexOf(T quarry)
{
    return 0;
}
/**
*  This removes the first insance of <code>quarry</code> from this
*  list; if no instance is found, it returns <code>false</code>.
*  @param quarry object to be removed to this list
*  @return <code>false</code> if <code>quarry</code> is not found
*  or the index where
*  the first instance of <code>quarry</code> is located.
*/
public boolean remove(T quarry)
{
    return false;
}
/**
*  This removes the object found at index <code>index</code>
*  @param quarry object to be removed to this list
*  @return <code>false</code> if <code>quarry</code> is not found
*  or the index where
*  the first instance of <code>quarry</code> is located.
*  @throws IndexOutOfBoundsException if an out-of-bounds index is passed
*/
public T remove(int index)
{
    return null;
}
/**
*  This removes all objects equal to <code>quarry</code> from this list.
*  @param quarry object to be extirpated from  this list
*  @return <code>false</code> if <code>quarry</code> is not found
*/
public boolean extirpate(T quarry)
{
    return false;
}
/**
*  This removes all elements from this list.
```

```
*/
public void clear()
{
}
/**
*  This computes the size of the list
*  @return the number of elements in this list
*/
public int size()
{
    return 0;
}
/**
*   This replaces the object at index <code>index</code> with <code>newItem</code>.
*   @param index the index at which we are performing the replacement
*   @param newItem the new item we are placing into the list
*   @return the item being replaced [the evictee]
*   @throws IndexOutOfBOundException if the <code>index</code> is out of bounds.
*/
public void set(int index, T newItem)
{

}
/**
*   This fetches the object at index <code>index</code>.
*   @param index the index from which we wish to retrieve an element.
*   @return the element at <code>index</code> in this list.
*   @throws IndexOutOfBOundException if the <code>index</code> is out of bounds.
*/
public T get(int index)
{
    return null;
}
/**
*   This checks for the presence of <code>quarry</code> in this list.
*   @param quarry the item we are searching for in this list
*   @return <code>true</code> if <code>quarry</code> is found in this list
*/
public boolean contains(T quarry)
{
    return false;
}
/**
 * This creates an iterator that walks throught this list in index order.
 * @return iterator for this list.
 */
```

```
    public Iterator<T> iterator()
    {
        return null;
    }
}
```

Use the array-based stack as a guide to help you get started. Implement the methods in this shell class and test them to see that you have a fully functioning stack. Notice that the client knows nothing about the internal workings of the stack; in particular the client programmer has no idea you are using links. He simply pushes, pop, and writes his application using your class.