JAVA: A Brief Tutorial

By:
Jeremiah A. Brown

For:
Y398 (Internship)
Prof. Hossein Hakimzadeh
Spring, 1999

Table of Contents

Introduction
Compiling and Running Java Classes
A Stack Class in Java
A GUI for Class Stack
Suggestions for Further Development
Resources and further reading
Introduction

This tutorial is for those who want to get a feel for some of the syntax and organization of the Java programming language and API by jumping in and building a working application. Familiarity with basic programming ideas is assumed (i.e., the object-oriented programming model, inheritance, exception handling, control structures, etc.) as well as some familiarity with the event-driven programming model commonly used in GUI development.

The tutorial is centered around the implementation of a basic ADT: a stack class. The class is designed as a re-usable module; then a GUI class (StackGUI) is developed and wrapped around the Stack class, to illustrate the use of some of the Abstract Windows Toolkit (AWT) components. In conclusion, some ideas for improving upon each of the classes (and encountering new topics in Java along the way) are suggested.

Compiling and running Java classes

Once the JDK is properly installed on your system, you can compile a Java class (which should be a file with a filename of the form *.java) by typing

```
javac <filename>
```

When the class is successfully compiled into a *.class file, you can run it using the Java Virtual Machine by typing

```
java <classname>
```

Note that <classname> is the name of the class, not the name of the file that the class is contained in. (In other words, leave off the .class extension.)

A Stack class in Java

Let's begin by examining the structure of our class, called class 'Stack'. The class will need the basic push and pop functions, to allow objects to be pushed on and popped off of the stack. We can represent the stack as a linked list of nodes; each node will contain the data object that was pushed onto the stack, and a pointer to the next node in the list.

It is also convenient to be able to determine the size of the stack (number of elements pushed on the stack) at any given time; to this end we can include an integer variable size in the Stack class, that will be modified whenever the stack object changes in size.

We also need to define the structure of the nodes that the stack will use. We can do this by using the notion of an inner class, which is simply a class whose definition is contained inside another class. The outer class will automatically have access to the inner class, so we don't need to worry about package statements or scope issues. Likewise, the methods of the inner class (if any) will be able to reference the structures and methods of the containing
### Stack.java

```java
import java.util.*;

public class Stack {
    private int size; // number of nodes in the stack
    private StackNode first; // handle to first element in the list

    // StackNode class (friendly access)
    class StackNode {
        Object element; // holds the data element for the node
        StackNode next; // handle to the next element in the list

        // constructors:
        // zero argument constructor:
        StackNode() { element = null; next = null; }
        // constructor with element initialization:
        StackNode(Object e) { element = e; next = null; }
        // constructor with element and next initialization:
        StackNode(Object e, StackNode n) {
            element = e; next = n;
        }
    }

    public void push(Object e) {
        // special case: pushing first node onto stack
        if(first == null)
            first = new StackNode(e);
        // normal case: pushing nth node
        else
            first = new StackNode(e, first);
    }
}
```

To run the code that is discussed below, you'll need to insert it into a text file called *Stack.java*. (The name of the file must always be the same as the name of the public class contained in it.) Here's what we have so far:
size++; //increase node count

public Object pop() {

    if(first == null)
        return null;

    //retrieve the first element in list
    Object e = first.element;

    //remove the first node from the list by pointing around it
    first = first.next;

    //decrement node count
    size--;

    //return the retrieved element
    return e;
}

The import statement at the very top of the listing is somewhat like the
#include statement used in C/C++; it allows the classes in the package java.util
to be referenced by the classes in Stack.java. (A package is just a collection
of classes.) Alternatively, we could leave off the import statement, and
reference classes in java.util by using the full package name with the class
name (i.e., java.util.Vector, to reference the Vector class.)

It's convenient (and trivial) to have several constructors defined for the inner
class StackNode. The variable first is a handle to the first node of the linked
list that represents the stack; it must be private, so that the integrity of the
list is guaranteed. The variable size should also be private, so that we can
depend on it to accurately represent the size of the stack. The keywords private
and public are used the same way that they are used in C/C++; anything marked
private will not be accessible outside of the class it is defined in (including
classes that inherit from it.) Things marked public are accessible everywhere.
The default access mode has no keyword associated with it, but behaves much like
the friendly access mode in C++; friendly access means that classes in the same
package can have access to the item. We're not organizing our classes into a
package structure, so this is irrelevant here.

One point of interest here is the use of the type Object as the data type in the
stack. Class Object is the at the very top of the(singly-rooted) class hierarchy
tree in the Java language; every class inherits from Object. Our use of it here
has a couple important ramifications. First, we can store heterogeneous objects
in our stack. We first push an integer object, and then a string, or a
character, and there would be no problems. The flip side to that is that when we
pop the objects back off, we must cast them to the proper type in order to do anything with them. If we try to cast an object to the wrong type, we'll get a runtime exception. So it makes sense to only store one type of object in our stack, to avoid confusion; but that type can be any type that we wish, as long as it's an object (as opposed to a primitive, such as int or char.) To use our Stack class with primitives, we'll have to encapsulate the primitive in a simple class. This is probably a small price to pay for the flexibility we get in using type Object for storage.

The above class has all the functionality of a normal stack, but it might be convenient (for debugging, at least) to be able to print out the contents of the stack. For this purpose, we'll add a function called elements(), which will return an Enumeration of the items on the stack. An Enumeration is essentially an iterator that points to some collection structure that contains a bunch of elements. The Enumeration has two methods: hasMoreElements(), which returns true if the collection has any elements in it; and nextElement(), which returns the next Object in the collection. If nextElement() is called and there are no more elements in the collection, a NoSuchElementException is thrown.

Since the Enumeration is such a convenient interface, we'll write a method that returns an Enumeration on the data elements in our stack. How do we do this, without a great deal of work? One way is to copy our data elements into another structure that already has a method that returns an Enumeration, and then call the appropriate method in that class to construct the Enumeration. One such class is Vector, which is a variable-sized array. The function below uses the Vector class's elements() function to accomplish our task:

```java
public Enumeration elements() {
    if(first == null)
        return new Vector().elements();
    //declare a vector for temporary use
    Vector v = new Vector();
    StackNode current = first; //point to first node in list
    //iterate through the list, copying the elements to the Vector v
    while(current != null) {
        v.addElement(current.element);
        current = current.next; //move to next node
    }
    //return an enumeration of the elements inserted into the vector
    return v.elements();
}
```

The function elements() is defined on all the container classes in
Java, including Vector, List, and Hashtable. By using this trick, we can have a relatively simple method that performs a very convenient and powerful task.

Now we'll define a method main() that will allow us to test the class, without involving any other classes. We'll push a couple items, print the stack out, pop a couple, and print it again:

```java
//a method main() to test the class
public static void main(String[] argv) {
    Stack s = new Stack();
    s.push(new String("First node"));
    s.push(new String("Second node"));
    s.push(new String("Third node"));
    //print contents of stack to screen
    Enumeration e = s.elements();
    while(e.hasMoreElements())
        System.out.println(e.nextElement());
    //pop a few things
    System.out.println("popped: " + s.pop());
    System.out.println("popped: " + s.pop());
    //print contents of stack to screen
    e = s.elements();
    while(e.hasMoreElements())
        System.out.println(e.nextElement());
}
```

Note that the method main is both public and static. This means there is only one "copy" of this function for all the objects of class Stack. Whenever an the class is invoked as a running program (i.e., using the Java Virtual Machine) it is this method that will be called.

Add the above two functions (elements() and main()) to the Stack class definition, and compile it. When you run the class, you'll see the following output:

Third node
Second node
First node
popped: Third node
popped: Second node

First node

As you can see, the Enumeration object returned by our elements() places the top node of the stack (most recently pushed object) at the front of the Enumeration, so that elements are returned by the Enumeration's nextElement() method in the same order that they would be popped off of the stack. Note that the Enumeration is a collection of the data elements that were pushed on the stack, not a collection of actual stack nodes.

While the pop() function returns null if called when the stack is already empty, we can cause the program to crash by continually calling pop and trying to print out the result. Specifically, we'll get a NullPointerException, if we execute the line

System.out.println("popped: " + s.pop());

too many times. This is because pop() returns a null value, which we're trying to convert to a String object (implicitly in the println call.) This is of course an impossible conversion. (To see this, add it two more times to the section in main under "pop a few things", and compile and run the code again.) This can happen if we use our stack class in another program, and forget to check the result of pop() for a null value. We can improve our stack class, using exceptions, to help avoid this kind of error. What we need to do is to define a special exception, StackEmptyException, that the method pop() can throw when it is called while the stack is empty. A programmer can ignore the value returned from pop(), but an exception cannot be ignored; it must be caught, or a compile-time error will be generated. This makes for fewer errors at run-time. Let's add the following class definition to our Stack class, as another inner class:

class StackEmptyException extends Exception {
    //constructors

    public StackEmptyException() {}
    public StackEmptyException(String msg) {
        super(msg);
    }
}

The line StackEmptyException extends Exception indicates that our class StackEmptyException inherits from the base class Exception. This means we'll have all the functionality of Java's Exception class in our new class (such as the ability to print stack traces, etc.) The call to the super method is a call to the base class constructor.

Now let's change pop() so that it throws a StackEmptyException if the stack is empty, instead of just returning a null value. We'll also have to declare in the
first line of method pop() that it can throw this exception, so that the compiler can be sure that whoever is calling pop() makes a provision for catching it. The new pop() method should look like this (new code is in bold print):

public Object pop() throws StackEmptyException {
    if(first == null)
        throw new StackEmptyException("Can't pop empty stack.");

    //retrieve the first element in list
    Object e = first.element;

    //remove the first node from the list, by pointing around it
    first = first.next;

    //decrement node count
    size--;

    //return the retrieved element
    return e;
}

Try to compile Stack.java now, and read the compiler's error message. It will complain that the method main(), which calls pop(), is not catching the StackEmptyException, or declaring it in its throws clause. (The throws clause is the part starting with the keyword throws, like the clause we just added to pop().) Now change main() so that it uses a try-catch block to contain the calls to pop(). This way, we don't have to check for a null value returned from pop(), we can just catch the exception and continue program execution. To illustrate the message-passing ability of the Exception class, we'll also print out the error message that pop() inserted in the StackEmptyException it throws. (This is a useful technique for debugging, but of course would be suppressed in a completed application.) The revised main() follows:

//a method main() to test the class

public static void main(String[] argv) {
    Stack s = new Stack();
    s.push(new String("First node"));
    s.push(new String("Second node"));
    s.push(new String("Third node"));
    //print contents of stack to screen
    Enumeration e = s.elements();
while(e.hasMoreElements())
    System.out.println(e.nextElement());

try{
    //pop a few things
    System.out.println("popped: " + s.pop());
    System.out.println("popped: " + s.pop());
    System.out.println("popped: " + s.pop());
    System.out.println("popped: " + s.pop());
}
}catch(StackEmptyException err) {
    System.out.println(err.getMessage());
}

//print contents of stack to screen

e = s.elements();
while(e.hasMoreElements())
    System.out.println(e.nextElement());

It is easy to imagine that we'll want to limit the number of things pushed on
the stack, as well. We can create another exception, StackFullException, that
push() will throw when called on a full stack. Let's also define a variable
maxSize that will define the arbitrary limit of the size of the stack. push() will
check to see whether size is equal to maxSize, and will throw a
StackFullException if it is. We'll then need to add a throws clause to push(),
and encapsulate the calls to push() in main() in a try block. Then we'll need a
way to specify the maxSize of our stack; it makes sense to do this in the
constructor. So we'll add a default constructor, and one that takes an integer
argument for maxSize. Let's define a sentinel value for maxSize, say -1, that
says there is no limit to the size of the stack (except that imposed by physical
memory, of course.) The next listing shows all of these changes to class Stack:

public class Stack {

    Stack() {}

    Stack(int maximumSize) { maxSize = maximumSize; }

    private int size; //number of nodes in the stack

    private int maxSize = -1;

    private StackNode first; //handle to first element in the list
StackNode class (friendly access)

class StackNode {
    Object element; //holds the data element for the node
    StackNode next; //handle to the next element in the list

    //constructors:
    //zero argument constructor:
    StackNode() { element = null; next = null; }
    //constructor with element initialization:
    StackNode(Object e) { element = e; next = null; }
    //constructor with element and next initialization:
    StackNode(Object e, StackNode n) {
        element = e; next = n;
    }
}

public void push(Object e) throws StackFullException {
    //if size >= maxSize (and maxSize is nonnegative)
    //the stack is already full
    if(maxSize >= 0 && size >= maxSize)
        throw new StackFullException
            ("Stack is full.");

    //special case: pushing first node onto stack
    if(first == null)
        first = new StackNode(e);

    //normal case: pushing nth node
    else
        first = new StackNode(e, first);

    size++; //increase node count
}

public Object pop() throws StackEmptyException {
if (first == null)
    throw new StackEmptyException("Can't pop empty stack.");

// retrieve the first element in list
Object e = first.element;

// remove the first node from the list, by pointing around it
first = first.next;

// decrement node count
size--;

// return the retrieved element
return e;

}

public Enumeration elements() {
    if (first == null)
        return new Vector().elements();

    // declare a vector for temporary use
    Vector v = new Vector();
    StackNode current = first; // point to first node in list

    // iterate through the list, copying the elements to the Vector v
    while (current != null) {
        v.addElement(current.element);
        current = current.next; // move to next node
    }

    // return an enumeration of the elements inserted into the vector
    return v.elements();
}

class StackEmptyException extends Exception {

    // constructors
    public StackEmptyException() {}
    public StackEmptyException(String msg) {

class StackFullException extends Exception {

    //constructors
    public StackFullException() {}
    public StackFullException(String msg) { super(msg); }

}  
//a method main() to test the class
public static void main(String[] argv) {
    Stack s = new Stack(2);
    try{
        //attempt to push items on the stack
        s.push(new String("First node"));
        s.push(new String("Second node"));
        s.push(new String("Third node"));
    }catch(StackFullException err) {
        System.out.println(err.getMessage());
    }
    //print contents of stack to screen
    Enumeration e = s.elements();
    while(e.hasMoreElements())
        System.out.println(e.nextElement());

    try{
        //pop a few things
        System.out.println("popped: " + s.pop());
        System.out.println("popped: " + s.pop());
        System.out.println("popped: " + s.pop());
    }
System.out.println("popped: "+s.pop());
}
catch(StackEmptyException err) {
System.out.println(err.getMessage());
}
//print contents of stack to screen
e = s.elements();
while(e.hasMoreElements())
System.out.println(e.nextElement());

In the above listing, the stack size is limited to 2. The third call to push() will generate a StackFullException. We now have a pretty robust stack class, that we can use in future Java programs. The next section uses the AWT (Abstract Window Toolkit) resources to encapsulate our Stack class inside an interactive GUI, which allows the user to manipulate a Stack object. This will cover the basics of windowed applications, and the Java 1.1 event model.

A GUI for class Stack

In order to keep our Stack class clean, we can make another class that provides a GUI and acts as an interface between a user and a Stack object. Let's call the class StackGUI. StackGUI will inherit from class Frame (which is a class in the java.awt package). The Frame class normally is the backbone of a windowed application. (For applets, we would inherit from the java.awt.Applet class.)

One of the first issues that confronts us is the layout of various controls on the main form. It makes sense to add all of the different buttons, textboxes, etc. when the StackGUI object is initialized, so we’ll do this in the constructor. Any controls that we need to have access to outside of the constructor should be made members of the class StackGUI, and declared separately. For objects that we only need to initialize, place on the form, and forget about (like a couple of the Labels), we can declare on the fly in the constructor, and make them anonymous (i.e., without any handle). The following are some initial declarations from class StackGUI, and the constructor:

```
public static final int STACK_SIZE = 10;
Stack myStack = new Stack(STACK_SIZE);
TextField[] stackNodes;
TextField lastPopped, toPush;
Button btnPush, btnPop, btnExit;
Label lblRightPanel = new Label("Stack contents:", Label.CENTER);
```
//constructor: set up the form

StackGUI() {

    // a Panel for the left half of the form, where the buttons and
    // editable text fields reside:
    Panel leftPanel = new Panel();
    leftPanel.setLayout(new GridLayout(10,2));

    // a Panel for the right half of the form, where the text fields that
    // display the contents of the stack reside:
    Panel rightPanel = new Panel();
    rightPanel.setLayout(new GridLayout(STACK_SIZE + 1, 1));

    // label for the "last item popped" textfield
    leftPanel.add(new Label("Last item popped:"));

    // construct and add text field for "last item popped"
    lastPopped = new TextField(30);
    lastPopped.setEditable(false); // user can't edit this field
    leftPanel.add(lastPopped);

    // label for the "item to push"
    leftPanel.add(new Label("Item to push:"));

    // text field to contain string to push
    toPush = new TextField(30);
    leftPanel.add(toPush);

    // add the push buttons, on their own panel
    Panel btnPanel = new Panel();
    btnPush = new Button("Push");
    btnPop = new Button("Pop");
btnExit = new Button("Exit");
btnPanel.add(btnPush);
btnPanel.add(btnPop);
btnPanel.add(btnExit);

//add button panel to left panel
leftPanel.add(btnPanel);

//add label to right panel, indicating its contents:
rightPanel.add(lblRightPanel);

//add 10 text fields to right Panel to hold the contents of the stack
stackNodes = new TextField[STACK_SIZE];
for(int i = 0; i < STACK_SIZE; i++) {
stackNodes[i] = new TextField(30);
stackNodes[i].setEditable(false);
rightPanel.add(stackNodes[i]);
stackNodes[i].setVisible(false); //initially invisible
}
setLayout(new GridLayout(1,2));
add(leftPanel);
add(rightPanel);
}

myStack is an object of the Stack class we just created. StackGUI will manipulate this, according to commands received from the user via the various controls that we've placed on the form. If you're familiar at all with graphics programming, some of the components (Buttons, etc.) above may look familiar. A TextField is an object that displays a single line of text; it can be set to editable or non-editable. We'll use an editable one to allow the user to enter a string to be pushed onto the stack (toPush), and a non-editable one to display the result of a pop from the stack (lastPopped). The Button class is just the conventional pushbutton. We'll use one for pushing things on the stack (btnPush), one for popping things off (btnPop), and one to exit the application (btnExit). A Label displays a single line of uneditable text.

Inside the constructor, the first thing we do is make a couple of Panels. These are invisible constructs that make it easier to arrange components on the form. Components such as buttons and textfields can be placed on the Panel, which is
then placed on the Frame. The reason this is so convenient is because each Panel object can have its own **layout manager**, which is the object that controls how components are arranged on a container component (such as a Frame or a Panel). The AWT provides several different layout managers (**GridLayout**, **BorderLayout**, **FlowLayout**, etc.) The default is the FlowLayout layout manager, which just places components in one long line, from left to right, in the order they are added to the container, until it runs out of room; then it just wraps around to the next line. We've used the GridLayout layout manager above, which divides the container into rows and columns. The number of rows and columns is specified in the GridLayout constructor. Components are placed in the grid starting at the upper left cell, and moving right to the end of the first row, then wrapping around to the second row, etc. (Somewhat like FlowLayout, except that only one component can occupy a cell, so we can control the spacing somewhat.)

The `stackNodes` variable is a handle to an array of TextField objects. We'll use these to represent items on the stack. As we create them and add them to the form, we set the display mode to invisible, so that they won't be drawn on the screen. When an item is pushed on the stack and we want to display one of these TextFields, we can just call the `setVisible()` method with a true argument, which will cause the object to be displayed.

Our Frame is now set up so that the user's interface components are on a Panel on the left side, while the TextFields that represent the stack are on another Panel on the right. Let's add a `main()` method to the class so that we can run it and see what the form looks like:

```java
public static void main (String[] argv) {

    //create an instance of StackGUI
    StackGUI me = new StackGUI();
    me.setSize(new Dimension(700, 500));
    me.show();

    }
```

Since `main` is a static method, we must declare a non-static instance of the class, in order to call methods like `show()` (which displays the Frame), `setSize()`, etc. Compile and run the StackGUI class. You should be able to click on pushbuttons, and enter text in the TextField (only the one that we set to editable!). Nothing is really functional yet; we have to add the code that will be executed when the user presses a button. Try to kill the application by closing the window (clicking on the button marked "X" in the upper right corner of the Frame.) Nothing happens! This is because that the "X" button is a push button like any other; we must write the code to cause the application to exit when that button is pushed.

The Java 1.1 event model is extremely object oriented, and very easy to use. Objects in the AWT such as Buttons and TextFields can "fire" (dispatch) certain events; we can add "listeners" to each object that listen for these events, and perform certain actions when they occur. For example, when you clicked on the button marked "X" on the Frame, a WindowClosing event was generated. We can add a listener to the Frame that responds to that event, and closes the application:
class WL extends WindowAdapter {

    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }

    (This class should be placed inside class StackGUI, but outside of anything else.) The method windowClosing will be called when a WindowClosing event is generated in the Frame, as soon as we add an instance of the class as a listener to the Frame (anywhere in the constructor):

    addWindowListener(new WL());

    Now if you compile and run StackGUI, you can click on the "X" button to exit the application.

    Now let's add the rest of the listeners. We'll need one for each of the push buttons. When the "Push" button is pressed, we'll need a class that pushes whatever string is in the "toPush" TextField onto the stack:

    class btnPushListener implements ActionListener {

        public void actionPerformed(ActionEvent e) {
            try{
                myStack.push(toPush.getText());
            }catch(Stack.StackFullException err) {} 

            displayStack();
        }

    Add an instance of this class to the btnPush button. We'll also need a class that pops a string off of the stack and writes it to the lastPopped TextField:

    class btnPopListener implements ActionListener {

        public void actionPerformed(ActionEvent e) {
            try{
                String s = (String)myStack.pop();
                displayStack();
                lastPopped.setText(s);
            }catch(Stack.StackEmptyException err) {
            }
        }
    }
An instance of btnPopListener should be added as an action listener to the btnPop button, in the constructor. Now the listener class for the "Exit" button; an instance of this must be added as an action listener to the btnExit button:

class btnExitListener implements ActionListener {
public void actionPerformed(ActionEvent e) {
    dispatchEvent(new WindowEvent(StackGUI.this, WindowEvent.WINDOW_CLOSING));
}
}

The btnExitListener class dispatches a WINDOW_CLOSING event to the main application window (which is an instance of StackGUI, which extends Frame; StackGUI.this references the instance of the StackGUI class that we're in.) The instance of the windowListener class WL that we added to the Frame (in the constructor) will respond to this event, and close the application. We could just call System.exit() from the btnExitListener class, but that would be in bad form; there should only be one exit point from the application, so that any cleanup that is necessary can be done in one central location. We don't really have any cleanup to do for StackGUI, but we'll stay in good form anyway.

Now, the function that does the real work in the GUI: displayStack(). This method will get an Enumeration of the elements in the Stack object that the StackGUI object is manipulating. Then it will set any unused TextFields at the top of the array to invisible, and fill in the rest with the contents of the stack. The topmost visible TextField will display the item most recently pushed onto the stack:

private void displayStack() {
    Enumeration e = myStack.elements();
    int i; //use this variable in two loops
    for(i = 0; i < STACK_SIZE - myStack.getSize(); i++)
        stackNodes[i].setVisible(false);
    for( ; i < STACK_SIZE; i++) {
        stackNodes[i].setText((String)e.nextElement());
        stackNodes[i].setVisible(true);
    }
}

Now we have everything we need in StackGUI to manipulate and display the Stack.
object contained inside. Here is a listing of the entire StackGUI class, in one
piece (don't forget the import statements):

// StackGUI.java

// A GUI wrapper to demonstrate the Stack class
import java.awt.*;
import java.awt.event.*;
import java.util.*;
class StackGUI extends Frame {
    public static final int STACK_SIZE = 10;
    Stack myStack = new Stack(STACK_SIZE);
    TextField[] stackNodes;
    TextField lastPopped, toPush;
    Button btnPush, btnPop, btnExit;
    Label lblRightPanel = new Label("Stack contents:", Label.CENTER);

    //constructor: set up the form
    StackGUI() {
        //a Panel for the left half of the form, where the buttons and
        //editable text fields reside:
        Panel leftPanel = new Panel();
        leftPanel.setLayout(new GridLayout(10,2));
        //a Panel for the right half of the form, where the text fields that
        //display the contents of the stack reside:
        Panel rightPanel = new Panel();
        rightPanel.setLayout(new GridLayout(STACK_SIZE + 1, 1));
        //label for the "last item popped" textfield
        leftPanel.add(new Label("Last item popped:"));

        //construct and add text field for "last item popped"
        lastPopped = new TextField(30);
        lastPopped.setEditable(false); //user can't edit this field
        leftPanel.add(lastPopped);
//label for the "item to push"
leftPanel.add(new Label("Item to push:"));

//text field to contain string to push
toPush = new TextField(30);
leftPanel.add(toPush);

//add the push buttons, on their own panel
Panel btnPanel = new Panel();
btnPush = new Button("Push");
btnPop = new Button("Pop");
btnExit = new Button("Exit");
btnPanel.add(btnPush);
btnPanel.add(btnPop);
btnPanel.add(btnExit);

//add button panel to left panel
leftPanel.add(btnPanel);

//add label to right panel, indicating its contents:
rightPanel.add(lblRightPanel);

//add 10 text fields to right Panel to hold the contents of the stack
stackNodes = new TextField[STACK_SIZE];
for(int i = 0; i < STACK_SIZE; i++) {
    stackNodes[i] = new TextField(30);
    stackNodes[i].setEditable(false);
    rightPanel.add(stackNodes[i]);
    stackNodes[i].setVisible(false); //initially invisible
}
setLayout(new GridLayout(1,2));
add(leftPanel);
add(rightPanel);
// add event handlers:
// a handler for the window closing event
addWindowListener(new WL());
// add listener to "Push" button, that pushes a string onto the stack
btnPush.addActionListener(new btnPushListener());
// add "Pop" button listener, that pops a string off the stack
btnPop.addActionListener(new btnPopListener());
// add "Exit" button listener, that dispatches a WINDOW_CLOSING event
// to the application window.
btnExit.addActionListener(new btnExitListener());
// display the stack (it will be empty)
displayStack();
}
class WL extends WindowAdapter {
public void windowClosing(WindowEvent e) {
    System.exit(0);
}
}
class btnPopListener implements ActionListener {
public void actionPerformed(ActionEvent e) {
    try{
        String s = (String)myStack.pop();
        displayStack();
        lastPopped.setText(s);
    }catch(Stack.StackEmptyException err) {
        System.out.println("Stack Empty.");
    }
}
}
class btnPushListener implements ActionListener {
public void actionPerformed(ActionEvent e) {
try {
    myStack.push(toPush.getText());
} catch (Stack.StackFullException err) {
}
displayStack();
}

// the actionPerformed method will dispatch a WINDOW_CLOSING event
// to the application window, which will cause the window to close and the
// application to exit. This is more graceful than simply calling
// "System.exit" from here.

class btnExitListener implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        dispatchEvent(new WindowEvent(StackGUI.this,
                                        WindowEvent.WINDOW_CLOSING));
    }
}

// updates the display of textfields representing the stack object
private void displayStack() {
    Enumeration e = myStack.elements();
    // the text fields were placed on the form with the lowest index at
    // the top. We want to represent the stack as growing from the bottom
    // of the form, so we'll place things in the array of text fields starting
    // with the highest index.
    // First, set all the unused nodes to invisible (based on size of stack):
    int i; // use this variable in two loops
    for(i = 0; i < STACK_SIZE - myStack.getSize(); i++)
        stackNodes[i].setVisible(false);
    for( ; i < STACK_SIZE; i++) {
        stackNodes[i].setText((String)e.nextElement());
        stackNodes[i].setVisible(true);
    }
}
public static void main(String[] argv) {
    //create an instance of StackGUI
    StackGUI me = new StackGUI();
    me.setSize(new Dimension(700, 500));
    me.show();
}

Suggestions for further development

The classes developed above can easily be extended and improved, especially the StackGUI class. Below are a couple of suggestions for areas of improvement. Undertaking these improvements will expose you to other areas of the Java language and API:

Develop a console-driven interface class for the Stack class. This could be something simple like a text-menu driven class, that prompts the user for strings to push on the stack, etc. This will necessitate investigation and use of some of the myriad input and output classes that Java has. In particular, take a look at the Reader, Writer, InputStream and OutputStream classes.

Make an inner class in class StackGUI that inherits from (extends) the Dialog class. Use this to cause a pop-up dialog to appear when the user to perform an illegal operation on the stack and causes an exception to be thrown (i.e., when our StackEmptyException or StackFullException are thrown.)

Make the application into an Applet. This is not hard to do, and can be fun. The StackGUI class can be made into an applet-only class, or into a class that can run either as a standalone application, or as an applet inside your web browser.

Resources and further reading

There are a host of great books out there on Java. The O'Reilly series is comprehensive and a great "black-book" resource. My favorite comprehensive tutorial on the Java language (and a source to which I referred constantly while writing this tutorial) is Bruce Eckel's "Thinking in Java" (www.BruceEckel.com.) The Java documentation that comes with the JDK (or is available at java.sun.com) is also very useful, and provides a complete specification for every class in the API.