JAVA NOTES

DATA STRUCTURES AND ALGORITHMS

Terry Marris July 2001

8 STACK IMPLEMENTATION - ARRAYS

8.1 LEARNING OUTCOMES

By the end of this lesson the student should be able to

- describe how an array may be used to implement the standard stack operations
- explain additional stack methods
- test stack methods
- use stacks to solve simple problems

8.2 INTRODUCTION

In the last lesson on the stack interface we saw that a stack is a linear data storage structure where items are added and removed from the same end. We described the properties of a stack. We described *what* the standard stack methods, *push()*, *pop()* and *peek()* do. Now we go on to describe *how* they do it. We shall implement the stack interface by using an array as the underlying data storage structure.

8.3 ARRAY IMPLEMENTATION

We shall need an array of objects and two instance variables, one to hold the number of elements in the array (capacity) and another to indicate the top of the stack.

```
  Object[] data;
  int capacity;
  int top;
```

The array is named *data*. We fix its *capacity* to (an arbitrary) five elements. Initially, the array is empty and *top* has the value -1.

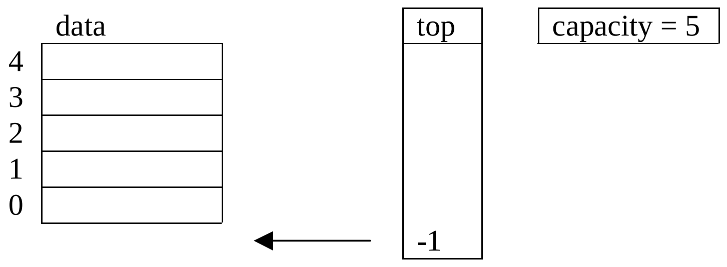


Figure 8.1 *An Empty Stack*

Instead of drawing the array horizontally, as we usually do, we have drawn it vertically with the first element (numbered zero) at the bottom.

Straight away we can write the implementation for *isEmpty()*. A stack is empty if *top == -1*. Or put another way, a stack is empty if *top* is less than zero.

```
  public boolean isEmpty()
  /*  Returns true if this stack contains no objects,
      false otherwise.
  */
  {
    return top < 0;
  }
```

A full stack might look like this.

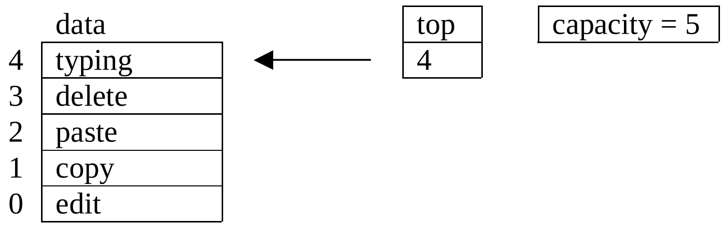


Figure 8.2 *A Full Stack*

A stack is full when *top + 1 == capacity*. We prefer to write *top + 1 >= capacity* because it is safer. The stack is certainly full if *top* is the same as *capacity*.

```
public boolean isFull()
/* Returns true if this stack cannot contain an additional
   object, false otherwise.
*/
{
  if (top + 1 >= capacity)
    return true;
  else
    return false;
}
```

To add an object to the stack we

> check to see if there is room
> if so, we increment top and store the object in data[top].

After adding *edit* to an empty stack the picture looks like this

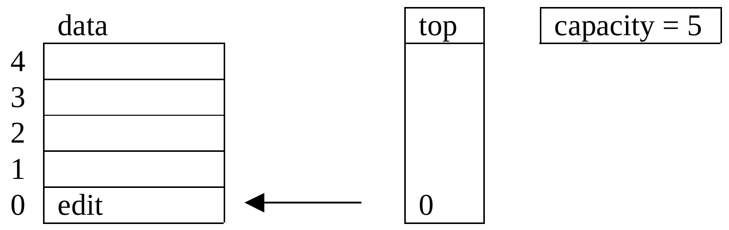


Figure 8.3 *A Stack with One Item*

A stack containing three items would look like this.

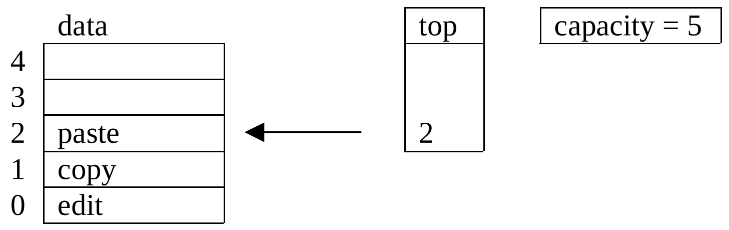


Figure 8.4 *A Stack with Three Items*

To remove an item we

> first check that the stack is not empty
> if so, we decrement top.

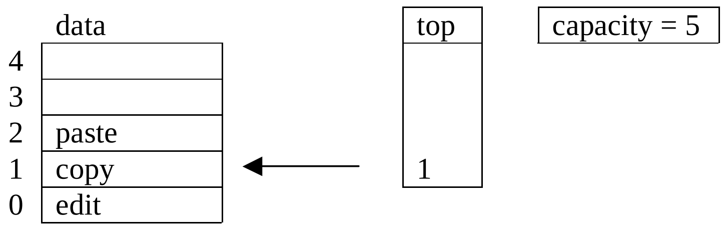


Figure 8.4 *A Stack with Two Items (After Removing One Item)*

Ok. *paste* is physically left in the stack. But it is *logically* removed because the next *push()* operation increments *top* and places the next object in *data[2]*, overwriting what was stored there before.

```
public String pop()
/* Removes the item at the top of this stack, if there is
   one, and returns success, otherwise returns failure.
*/
{
  if (isEmpty())
    return "failure - stack empty";
  top--;
  return "success";
}
```

The *peek()* operation just returns the object at location *top* in the array. In Figure 8.4 *copy* would be returned.

```
public Object peek()
/* Returns the item at the top of this stack, if there is
   one, otherwise returns null.
*/
{
  if (isEmpty())
    return null;
  return data[top];
}
```

The complete implementation of the *Stack* interface is shown below. Notice that

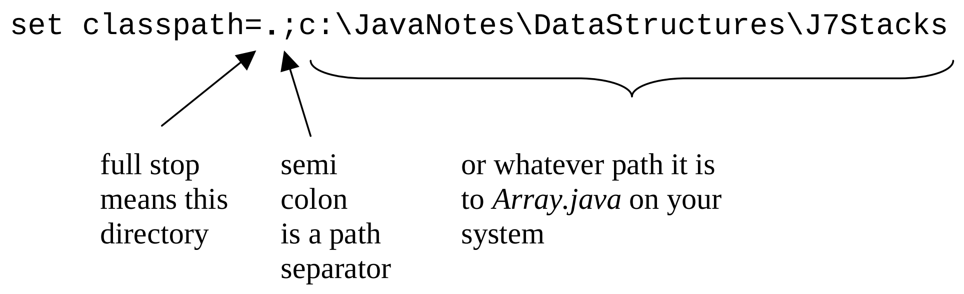
we have documented each method with a short description of WHAT it does
we have provided additional methods, *isFull()*, *size()* and *getCapacity()*
we have provided two constructors (an interface cannot have constructors because it cannot be instantiated)

```
/* ArrayStack.java
   Terry Marris  19 July 2001
*/

public class ArrayStack implements Stack {
  /* Implements the standard stack operations push,
     pop, peek and isEmpty().  In addition implements
     constructors, isFull(), getCapacity() and size().
  */

  private Object[] data;
  private int capacity;
  private int top;


  public ArrayStack(int capacity)
  /* Initialises a new stack with the given capacity.
  */
  {
    capacity = Math.abs(capacity);  // ensure capacity >= 0
    this.capacity = capacity;
    data = new Object[capacity];
    top = -1;
  }


  public ArrayStack()
  /* Initialises a new stack with a capacity of 10 objects.
  */
  {
    this(10);
  }


  public boolean isEmpty()
  /*  Returns true if this stack contains no objects,
      false otherwise.
  */
  {
    return top < 0;
  }
```

```
public boolean isFull()
/* Returns true if this stack cannot contain an additional
   object, false otherwise.
*/
{
  if (top + 1 >= capacity)
    return true;
  else
    return false;
}


public String push(Object obj)
/* Places the given object on this stack, if there is room,
   and returns success,
   otherwise returns failure.
*/
{
  if (isFull())
    return "failure - stack full";
  top++;
  data[top] = obj;
  return "success";
}


public String pop()
/* Removes the item at the top of this stack, if there is
   one, and returns success, otherwise returns failure.
*/
{
  if (isEmpty())
    return "failure - stack empty";
  top--;
  return "success";
}


public Object peek()
/* Returns the item at the top of this stack, if there is
   one, otherwise returns null.
*/
{
  if (isEmpty())
    return null;
  return data[top];
}
```

```
  public int getCapacity()
  /* Returns the maximum number of objects this stack can
     contain.
  */
  {
    return capacity;
  }


  public int size()
  /* Returns the number of objects currently held on this
     stack.
  */
  {
    return top + 1;
  }
}
```

Important note: Your *Stack* interface may be in a different directory to the one that contains your *ArrayStack* class. If you compile *ArrayStack.java* and the compiler reports *Class Stack not found* you must tell the compiler where to look for your *Array* interface by using the *set classpath* command. For example, at a DOS prompt you might enter



```
     set classpath=.;c:\JavaNotes\DataStructures\J7Stacks
```

full stop means this directory
semi colon is a path separator
or whatever path it is to *Array.java* on your system

Notice that there is no space before and after the = symbol.

8.4 TESTING THE ARRAY STACK

The test program shown below creates a stack with a capacity of five, pushes five objects onto a stack, and then empties the stack item by item.

```
/* TestArrayStack.java
   Terry Marris  19 July 2001
*/

public class TestArrayStack {
  public static void main(String[] s)
  {
    ArrayStack stack = new ArrayStack(5);
    System.out.println("A new stack is an empty stack ... " +
                       stack.isEmpty());

    System.out.println("Adding five strings ...");
    System.out.println("if: " +
                       stack.push(new String("if")));
    System.out.println("else: " +
                       stack.push(new String("else")));
    System.out.println("switch: " +
                       stack.push(new String("switch")));
    System.out.println("case: " +
                       stack.push(new String("case")));
    System.out.println("default: " +
                       stack.push(new String("default")));

    System.out.println("The stack is now full ..." +
                       stack.isFull());

    System.out.println("Adding a string to a full stack ...");
    System.out.println("while: " +
                       stack.push(new String("while")));

    System.out.println("Emptying the stack item by item ...");
    while (!stack.isEmpty()) {
      System.out.println(stack.peek());
      stack.pop();
    }
  }
}
```

Output

```
A new stack is an empty stack ... true
Adding five strings to the stack ...
if: success
else: success
switch: success
case: success
default: success
The stack is now full ...true
Adding a string to a full stack ...
while: failure - stack full
Emptying the stack item by item ...
default
case
switch
else
if
```

Notice that the output is in reverse order of the input to the stack.

8.5 REVIEW

8.6 FURTHER READING

In the next lesson we see how to implement a stack without using an array.

8.7 EXERCISES

1 Test the methods *isFull()*, *getCapacity()* and *size()*.

2 Write and test the method *boolean isBalanced(String arithmeticExpression)* that returns true if the brackets in the given arithmetic expression are both balanced and well formed. for example

(a + b)	balanced and well-formed
(a + b	unbalanced
a + b)	unbalanced
(a + b)) * ((c + d)	balanced (equal numbers of (and), but not well-formed