12: THE STACK DATA STRUCTURE

Introduction 1
Operations 1
Static and dynamic data structures 5
The Stack class 5
EXERCISE: Creating the Stack class 8
Using a Stack 9
EXERCISE: Using a Stack 11
EXERCISE: Storing other types of data 12
EXERCISE: An practical application of the *Stack* class 13
EXERCISE: Algorithms using stacks 15

Introduction

In the following chapters we will look at some examples of abstract data structures. These structures store and access data in different ways which are useful in different applications. In all cases the structures follow the **principle of data abstraction** (the data representation can be inspected and updated only by the abstract data type's operations). Also, the algorithms used to implement the operations **do not depend on the type of data** to be stored.

A **stack** is a limited version of an array. New elements, or **nodes** as they are often called, can be added to a stack and removed from a stack only from one end. For this reason, a stack is referred to as a LIFO structure (Last-In First-Out).

Stacks have many applications. For example, as processor executes a program, when a function call is made, the called function must know how to return back to the program, so the current address of program execution is pushed onto a stack. Once the function is finished, the address that was saved is removed from the stack, and execution of the program resumes. If a series of function calls occur, the successive return values are pushed onto the stack in LIFO order so that each function can return back to calling program. Stacks support recursive function calls in the same manner as conventional nonrecursive calls.

Stacks are also used by compilers in the process of evaluating expressions and generating machine language code. They are also used to store return addresses in a chain of method calls during execution of a program.

Operations

An abstract data type (ADT) consists of a data structure and a set of **primitive operations**. The main primitives of a stack are known as:

Push adds a new node
Pop removes a node

Additional primitives can be defined:

IsEmpty reports whether the stack is empty
IsFull reports whether the stack is full
Initialise creates/initialises the stack
Destroy deletes the contents of the stack (may be implemented by re-initialising the stack)

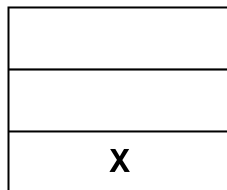
Initialise

Creates the structure – i.e. ensures that the structure exists but contains no elements

e.g. ***Initialise(S)*** creates a new empty stack named S

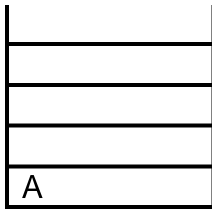
Push

e.g. ***Push(X,S)*** adds the value X to the TOP of stack S

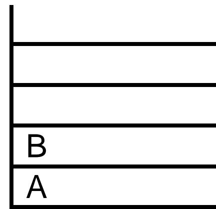


Pop
e.g. ***Pop(S)*** removes the TOP node and returns its value

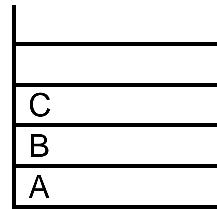
Example



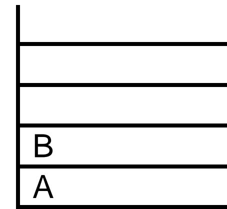
s.push('A');



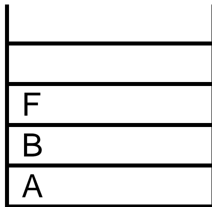
s.push('B');



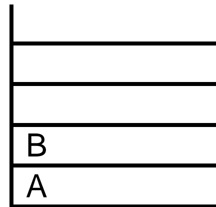
s.push('C');



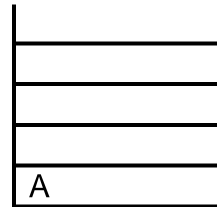
s.pop();
returns C



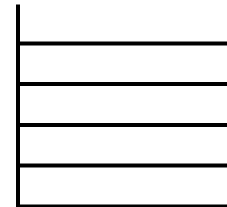
s.push('F');



s.pop();
returns F



s.pop();
returns B



s.pop();
returns A

We could try the same example with actual values for A, B and C.

A = 1 B = 2 C = 3

EXERCISE: Stack operations

1. What would the state of a stack be after the following operations:

    create stack
    push A onto stack
    push F onto stack
    push X onto stack
    pop item from stack
    push B onto stack
    pop item from stack
    pop item from stack

2. Show the state of the stack and the value of each variable after execution of each of the following statements:

    A = 5 B = 3 C = 7

(a) create stack
push A onto stack
push C*C onto stack
pop item from stack and store in B
push B+A onto stack
pop item from stack and store in A
pop item from stack and store in B

(b) create stack
push B onto stack
push C onto stack
push A onto stack
A = B * C
push A+C onto stack
pop item from stack and store in A
pop item from stack and store in B
pop item from stack and store in C

Stack Implementation

The Java Collections Framework includes a set of ready made data structure classes, including a *Stack* class. However, you will create your own stack class in order to learn how a stack is implemented. Your class will be a bit simpler than the Collections Framework one but it will do essentially the same job.

Static and dynamic data structures

A stack can be stored in:

- a static data structure

  OR

- a dynamic data structure

Static data structures

These define collections of data which are fixed in size when the program is compiled. An **array** is a static data structure.

Dynamic data structures

These define collections of data which are variable in size and structure. They are created as the program executes, and grow and shrink to accommodate the data being stored.

The Stack class

This Stack class stores data in an array (static structure). The array reference type is **Object[]** which means that it can contain **any kind of Java object**. This is because of **polymorphism** – every Java class is a subclass of *Object*.

The **constructor** creates a new array with its size specified as a parameter. The constructor does the job of the **Initialise** primitive described before.

An instance variable **total** keeps track of how many items are stored in the stack. This changes when items are added or removed. The stack is full when *total* is the same as the size of the array.

Stack
- stack: Object ([]) - total: int
+ Stack(int) + push(Object) : boolean + pop() : Object + isEmpty() : boolean + isFull() : boolean + getItem(int) : Object + getTotal() : int

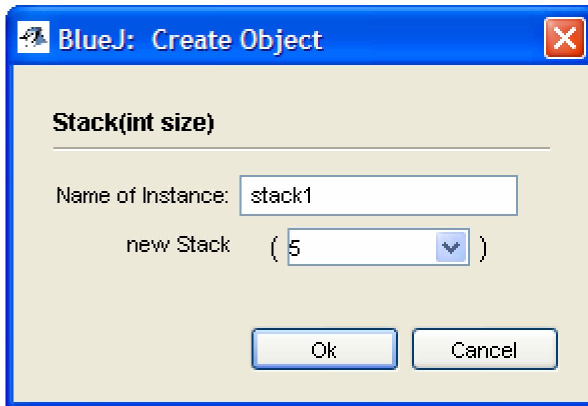
```
/**
 * class Stack
 *
 * @author Jim
 * @version 1.0
 */
public class Stack {
    private Object[] stack ;
    private int total;    // to track number of items

    public Stack(int size) {
        stack = new Object[size];  // create array
        total = 0;     // set number of items to zero
    }

    /**
     * add an item to the array
     */
    public boolean push(Object obj) {
        if ( isFull() == false)  // checks if space in stack
        {
            stack[total] = obj;   // add item
            total++;              // increment item counter
            return true;          // to indicate success
        }
        else {
            return false;       // to indicate failure
        }
    }

    /**
     * remove an item by obeying LIFO rule
     */
    public Object pop() {
        if (isEmpty() == false)   // check stack is not empty
        {
            // reduce counter by one
            Object obj = stack[total-1];  // get last item
            stack[total-1]= null;  // remove item from array
            total--;               // update total
            return obj;            // return item
        }
        else {
            return null;       // to indicate failure
        }
    }
```

```
    /**
     * checks if array is empty
     */
    public boolean isEmpty() {
        if (total ==0) {
            return true;
        }
        else {
            return false;
        }
    }

    /**
     * checks if array is full
     */
    public boolean isFull() {
        if (total ==stack.length) {
            return true;
        }
        else {
            return false;
        }
    }


    /**
     * returns the item at index i
     */
    public Object getItem(int i) {
        return stack[i-1];      // ith item at position i-1
    }


    /**
     * return the number of items in the array
     */
    public int getTotal() {
        return total;
    }

}
```

EXERCISE: Creating the Stack class

Create a new BlueJ project called *stacks* and create a new class *Stack* using the above code.

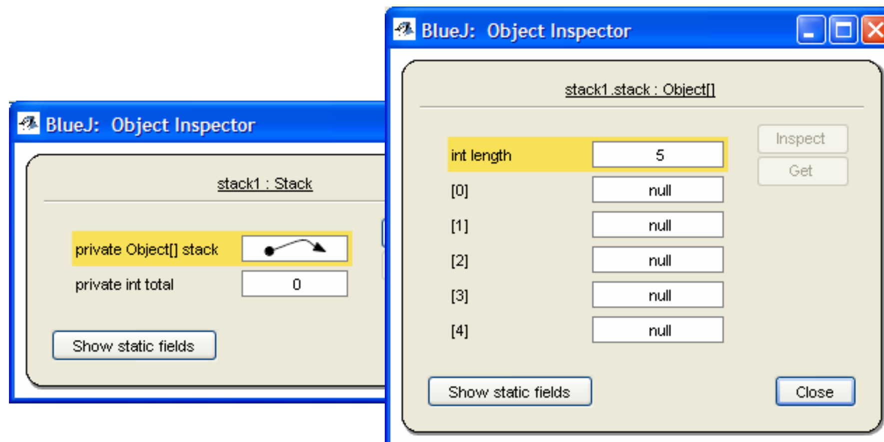
Create a new instance of *Stack* with size 5.



Inspect the *Stack* instance.

What variables does it have?

Click to select *private Object[] stack* and click the **Inspect** button. You should see an Object Inspector window like this:



Describe what is being shown here.

Using a Stack

Data structure classes are intended to be used in programs as utility classes which contain the data the program needs to work with. To use the *Stack* class, you need to know how to write code to call the *Stack* operations, for example to add data to the *Stack*.

Remember that the *Stack* can hold any kind of data. The following test class shows how to use a *Stack* to hold *Integer* objects. Calls to *Stack* operations are shown in bold.

```
/**
 * class StackTester
 *
 * @author Jim
 * @version 1.0
 */
public class StackTester {
    private Stack stack;

    public StackTester(){
        stack = new Stack(10);
    }

    public StackTester(Stack stack){
        this.stack = stack;
    }

    /**
     * push item onto stack
     */
    public void pushNumber(int num) {
        boolean ok = stack.push(new Integer(num));
        if (!ok)
            System.out.println("Push unsuccessful");
        else
            System.out.println("Push successful");
    }

    /**
     * pop number from stack
     */
    public void popNumber() {
        Integer result = (Integer) stack.pop();
        if (result!=null)
            System.out.println("Number is :" +
               result.intValue());
```

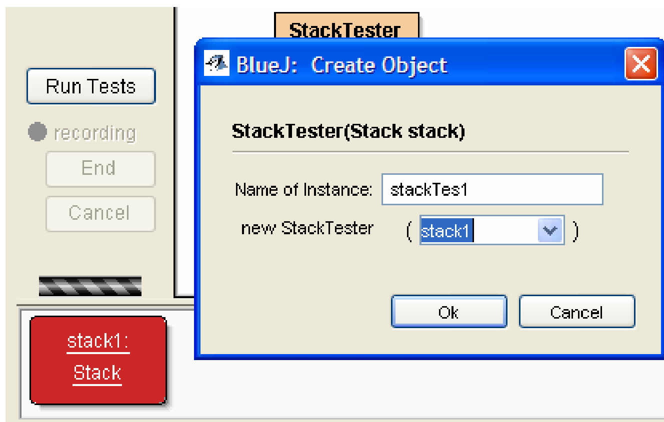
```
        else
            System.out.println("Pop unsuccessful");
    }

    /**
     * check if stack is empty
     */
    public void checkIfEmpty() {
        if (stack.isEmpty())
            System.out.println("Stack empty");
        else
            System.out.println("Stack is not empty");
    }

    /**
     * check if stack is full
     */
    public void checkIfFull() {
        if (stack.isFull())
            System.out.println("Stack full");
        else
            System.out.println("Stack is not full");
    }

    /**
     * list the numbers in stack
     */
    public void listNumbersInStack() {
        if (stack.isEmpty()) {
            System.out.println("Stack empty");
        }
        else {
            System.out.println("Numbers in stack are: ");
            System.out.println();
            for (int i=stack.getTotal(); i>=1; i--) {
                System.out.println(stack.getItem(i));
            }
            System.out.println();
        }
    }
}
```

EXERCISE: Using a Stack

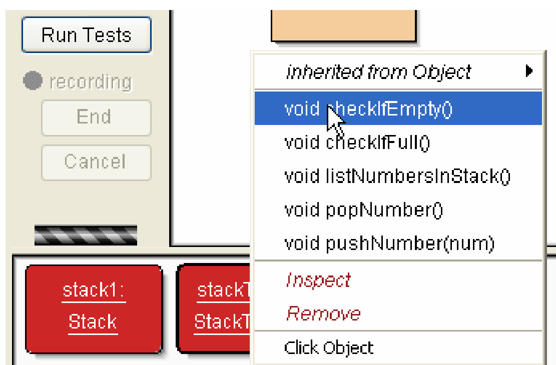
Add a new class *StackTester* to your stacks project using the above code.

Create a new instance of *Stack* with size 5.

Create a new instance of *StackTester* and select your *Stack* instance in the object bench as the parameter in the constructor. This means that you will be testing the *Stack* you created in the previous step.



Call the *checkIfEmpty* method of your *StackTester*.



What was the result?

Call the *pushNumber* method of your *StackTester* to add the number 5 to the stack. Inspect the *Stack* instance. Repeat this to add the number 7 and repeat again to add the number 2.

What result would you expect if you pop from the Stack?

Call the *popNumber* method of your *StackTester* and check that you got the correct result.

Call the *pushNumber* method of your *StackTester* to add the number 9 to the stack.

What do you expect the contents of the Stack to be now?

Inspect your *Stack* to check that you are correct. You will have to inspect each array element to see its integer value.

Call the methods of *StackTester* as needed to add and remove more numbers and check what happens when the *Stack* is full and when it is empty.

EXERCISE: Storing other types of data

Modify the *StackTester* class to store ***String*** objects in a *Stack* instead of *Integer* objects, and test in a similar way to the above.

You should **not** have to change the *Stack* class at all**.**

EXERCISE: An practical application of the *Stack* class

Compilers make use of stack structures. This example shows a simple illustration of the way a stack can be used to check the syntax of a program. It uses the Stack class you have created.

In the example, a *Stack* is used to check that the braces { and } used to mark out code blocks in a Java source file are matched – i.e. that there are the same number of { as }.

- The source file is read character by character.
- Each time a { is read an object (any object, it doesn't matter what) is pushed onto the stack.
- Each time a } is read an object is popped from the stack.
- If the stack is empty when a } is read then there must be a missing { .
- If the stack is not empty at the end of the file then there must be a missing } .

Add the following class *BracketChecker* to your *stacks* project:

```
import java.io.*;
import java.net.URL;

public class BracketChecker
{

    private Stack myStack = new Stack(100);

    public void check() throws IOException {
        // opens file in project directory
        URL url = getClass().getClassLoader().
               getResource("Track.java");
        if (url == null)
            throw new IOException("File not found");
        InputStream in = url.openStream();

        int i;
        char c;

        // read each character in the file
        // add a new object to the stack every time an
        // opening brace is read
        // remove an object every time a closing brace is
        // read
       while ((i = in.read()) != -1) {
           c = (char)i;
```

```
             // print out the current character
             System.out.print(c);

            // if character is closing brace, the stack
            // should not be empty
            if (c == '}') {
                if (myStack.isEmpty()) System.out.println("\n
                     ***** Error: missing {  *****");

                 // remove top object of stack
                 else myStack.pop();
             }
             else {
                 if (c == '{')
                      myStack.push(new Object());
             }
         }
         // stack should end up empty if braces balance
         if (!myStack.isEmpty()) System.out.println("\n *****
                Error: missing }  *****");

         in.close();
    }
}
```

Add another Java class to your project as a test file. You can cut and paste code from a previous exercise. You will need to specify the name in *BracketChecker* – the code above assumes the file is called *Track.java*. *(The rather complicated looking code to open the file is necessary to get at a file in the project directory.).*

Compile both classes.

Create an instance of *BracketChecker* and call its *check*() method.

What output do you get?

Remove a **{** in your test file and try to compile. You should get an error. Create a new instance of *BracketChecker* and call its *check()* method.

What output do you get?

Replace the missing **{** in your test file and remove a **}** Try to compile. You should get an error. Create a new instance of CheckBraces and call its check() method.

What output do you get?

EXERCISE: Algorithms using stacks

A stack is a natural structure for reversing a list as it will output values in the reverse order in which they are stored.

(a) Design an algorithm, using a stack, to read five characters from a keyboard and display them in reverse order.

(b) Design an algorithm, using a stack, which will convert a decimal number to binary.

(c) Design an algorithm, using a stack, which will test whether an input string is palindromic or not.

A palindrome is a word / sentence which reads the same backwards as forward.

e.g. *level*

A man, a plan, a canal, Panama