# Classes and Static Data Structures

**This first exercise shows how to interface a data structure with a GUI. The ideas of this exercise are general and can be applied to other data structures.**

**We are going to** d**evelop a Stack based Application for Jobs.**

We assume that a Job has an ID, a short text description, an employee to complete the job

Employee is a Person has a name, a specialism, a salary (in K)

Application should allow jobs to be added, removed from the Stack, with a count of how many jobs are on the Stack.

Start by creating a **Windows Forms Application** and add the classes below. If you do not remember how to create a Windows Forms Application, please revise the Lab of Week 1. Revise also the slides of the lecture on Classes and OOP in C#.

class Person : IComparable

{

private int age;

private string name;

public Person(string name) //constructor with one argument

{

this.name = name;

age = 18; //default age

}

public string Name

{

get { return name; }

set { name = value; }

}

public int CompareTo(Object obj) //implementation of CompareTo

{ // for IComparable

Person other = (Person)obj;

return Name.CompareTo(other.Name);

}

}

Add an Employee class, which inherits from Person

class Employee : Person

{

private string specialism;

private int salary; //in Thousands K

public Employee(string name, string specialism, int salary): base(name)

{

this.specialism = specialism;

this.salary = salary;

}

}

Add a Job Class

class Job

{

string id, description;

Employee person;

public Job(Employee person, string id, string desc)

{

this.person = person;

this.id = id;

this.description = desc;

}

public string ToString()

{

return id + ":" + description + ", employee:" + person.Name;

}

}

**Now the next step is to amend the Stack Class to handle Jobs (instead of int).**

Add and implement a **StackJob** class. Start from stack class IntStack (below). The class IntStack is for a stack of int.

We need to amend the code to store Jobs instead of int.

Looking for type int – where should we switch to Job?

Also, add in the StackJob class a public method Count() which returns the number of elements present in the stack.

class IntStack

{

private const int maxsize = 10;

private int top = 0;

private int[] array = new int[maxsize];

public void Push(int value)

{

array[top++] = value;

}

public int Pop()

{

return array[--top];

}

public int Peek()

{

return array[top - 1];

}

public bool IsEmpty()

{

return top ==0;

}

public bool IsFull()

{

return top == maxsize;

}

public string Print()

{

StringBuilder output = new StringBuilder();

for (int i = top - 1; i >= 0; i--)

output.Append(array[i] + Environment.NewLine);

return output.ToString();

}

}

Once we have added and implemented the class StackJob , we can interface it with the form in this way

public partial class Form1 : Form

{

private StackJob jobs;

public Form1()

{

jobs = new StackJob(); //create empty Stack

InitializeComponent();

}

We can now access the object jobs from the Form.

Add appropriate textboxes to the GUI (salaryTextBox, nameTextBox, specilaismTextBox, jobIDtextBox, jobDescTextBox) a label (countLabel ) to read the information and show the output, and buttons to Add a job on the stack (call it AddButton) and Remove a Job from the Stack .

We can add code for an event that reads info of a Job from the textboxes, and push the created Job object into the stack.

private void AddButton\_Click(object sender, EventArgs e)

{

//create an Employee object

int salaryK;

if (!int.TryParse(salaryTextBox.Text, out salaryK))

salaryK = 30; //default value

Employee person = new Employee(nameTextBox.Text, specilaismTextBox.Text, salaryK);

//create a job object

Job job = new Job(person, jobIDtextBox.Text, jobDescTextBox.Text);

//add job to Stack

jobs.Push(job);

countLabel.Text = "Count:" + jobs.Count();

}

We can add code that removes a Job from the stack (add an opportune button removeButton and textbox outTextBox)

private void removeButton\_Click(object sender, EventArgs e)

{

Job nextJob = jobs.Pop();

outTextBox.Text = nextJob.ToString();

countLabel.Text = "Count:" + jobs.Count();

}

## **Exercises on Stacks and Queues**

Complete the following exercises (up to exercise 6) to review some key concepts on stacks and queues. They will be very useful to complete the assessed exercises.

Exercises 7 - 10 are extra tasks, more advanced, useful to complement your knowledge on this topic and for future job interviews.

## Exercise 1

Create a Console Application for C#. The following code in the main) provides the outline of a simple stack class and a command line test program.

class Program

{

static void Main(string[] args)

{

IntStack mystack = new IntStack();

mystack.Push(10);

System.Console.WriteLine(mystack.Pop());

mystack.Push(20);

mystack.Push(30);

mystack.Push(40);

System.Console.WriteLine(mystack.Pop());

System.Console.WriteLine(mystack.Pop());

System.Console.WriteLine(mystack.Pop());

System.Console.ReadKey();

}

}

You will need to add a new class to your project in a file called IntStack.cs (code below).

class IntStack

{

private const int maxsize = 10;

private int top = 0;

private int[] array = new int[maxsize];

public void Push(int value)

{

array[top++] = value;

}

public int Pop()

{

return array[--top];

}

public int Peek()

{

return array[top - 1];

}

public bool IsEmpty()

{

return top ==0;

}

public bool IsFull()

{

return top == maxsize;

}

public string Print()

{

StringBuilder output = new StringBuilder();

for (int i = top - 1; i >= 0; i--)

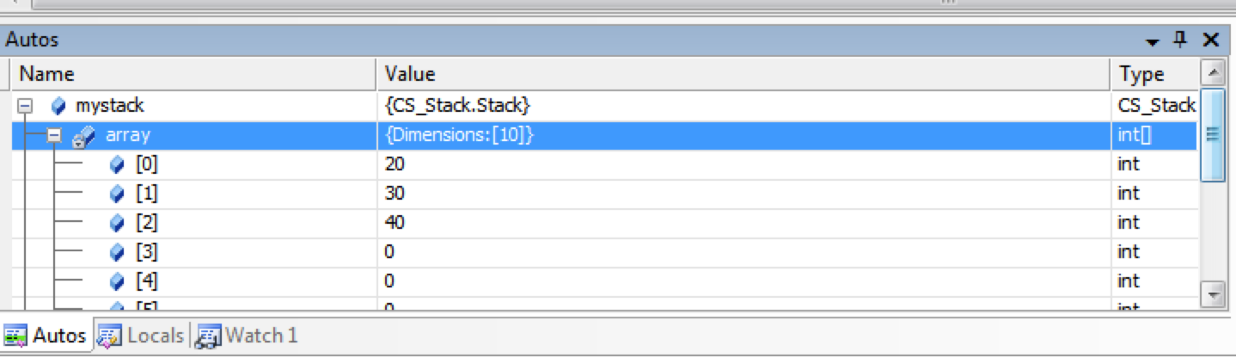
output.Append(array[i] + Environment.NewLine);

return output.ToString();

}

}

Use the debugger variables window to examine the values in the array.



### Exercise 2

In the test program (main) write a test program that :-

Uses a while loop to fill the stack with Random numbers.

Uses a separate while loop to pop each value and print it to the screen.

Random numbers can be generated in this way

Random randgen = new Random();

randgen.Next(100) // will produce a random number between 0-100

## Exercise 3

In a new Console Application, use the outline class below create a Queue class in file called IntQueue.cs and a main function to test your Queue. The class IntQueue needs to be completed following the materials seen in the lecture.

class IntQueue

{

private readonly int maxsize = 10;

private int[] store;

private int head = 0;

private int tail = 0;

private int numItems;

public IntQueue()

{

store = new int[maxsize];

}

public IntQueue(int size)

{

maxsize = size;

store = new int[maxsize];

}

public void Enqueue(int value)

{

// insert code

}

public int Dequeue()

{

// insert code

}

public int Peek()

{

// insert code

}

public bool IsEmpty()

{

return //insert code

}

public bool IsFull()

{

return // insert code

}

} //class

## Exercise 4

Create a new public function that returns the number of items in the Queue.

## Exercise 5

Create a new public function within your Queue class that returns a string containing the contents of the Queue.

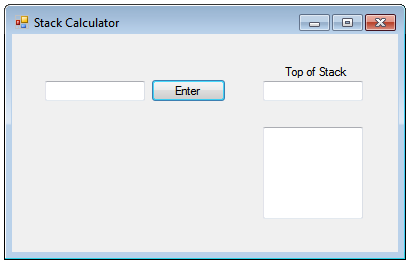
## Exercise 6

Add functionality to your Queue class to make it a circular Queue (see lecture slides).

## Exercise 7 (Advanced) Palindromes and Stack (often given at job interviews)

Use your implemented Stack class to write a program that checks if an input string is palindrome.

## Exercise 8



Download the RPN calculator (StackCalculator) that contains the skeleton (unfinished) code to produce the program shown above. You will need to replace the empty Stack class with your implementation done in exercise 2. Once you have replaced the empty Stack class with your implementation use the code to test the basic arithmetic (+ and -) in RPN notation (see below for a description of RPN).

## Exercise 9

Extend the program to implement the multiplication ( \* ) and division ( / ) operations as well.

## Exercise 10

In an RPN calculator an exchange function that exchanges the top two elements of the stack is useful. Extend the stack class to include the exchange function and extend the calculator so that when the user enter <> the top two elements are exchanged.

RPN (Reverse Polish Notation) is a form of postfix notation that is simpler to implement on computers than the regular infix notation we generally use. RPN has the advantage that Parenthesis are not required.

For instance,

the usual infix notation (5 -3)\*2 can be expressed in RPN as 5 3 - 2 \*

Interestingly, HP sold many RPN calculators that were extremely popular in the 70’s. Even now HP sell a iPhone app which replicates the look of one of these early calculators for nearly 20GBP.

http://www.hpmuseum.org/rpn.htm