# Review: Classes (public/private, methods, data fields), Arrays

**The goal for this exercise** is to get some practice creating a more specialized subclass, as well as to practice implementing a stack of simple types.

In this lesson, we're going to both implement a Stack, as well as examine one use of object-oriented programming (specifically, inheritance) : using inheritance to specialize an existing class.

For this exercise, you need to create a subclass of the SmartArray class, named StackOfInts. Your StackOfInts class will need to support the Push, Pop, Peek, IsEmpty and getSize methods.

In order to do that, you will need to add a method to the SmartArray class – getSize.

All of the methods for both the SmartArray, and StackOfInts classes, are described in more detail at the end of this document.

For this exercise, you are welcome to (and encouraged to) simply reuse your code for the SmartArray class from a prior exercise.

**What you need to do for this exercise:**

1. For this exercise, you need to implement the StackOfInts class. A more detailed description of this class is included at the end of this document. This class should be found in the **Student\_Answers.cs** file in a project named something like **03\_PCE\_StudentCode**.
   1. As part of this, you will need to add the getSize method to the SmartArray class.
2. Once you’ve completed the above task, you should run the tests in the Test\_Stack\_SmartArray\_Basic class and the Test\_StackOfInts class. They should all pass at this point, and if not, then you should fix your program so that those tests do pass. You can run the tests using the NUnit support in the provided starter project. You should feel free to supplement that code with your own test cases if you wish, but you are neither required nor expected to.  
   1. The Test\_Stack\_SmartArray\_Basic class serves to make sure that your StackOfInts still does everything that a basic SmartArray does. It is expected that your StackOfInts will be able to pass these tests simply by virtue of inheriting methods from the SmartArray base class.  
      1. Note that this means that your StackOfInts class MUST inherit from the SmartArray class in order for these tests to even compile!
   2. The Test\_StackOfInts class serves to check new, Stack-specific functionality that you’ve implemented in this exercise.

|  |  |  |  |
| --- | --- | --- | --- |
| **SmartArray Data & Methods** | | | |
| Data Field Name | | Type | Description: |
| rgNums | | Array of ints | A reference to an array of integers. |
| Note: all data fields should be marked **private** | | | |
| Unchanged Methods: | | | |
| These are unchanged from the prior implementation:  Default constructor,  GetAtIndex,  PrintAllElements,  Find,  SetAtIndex | | | |
| New Method Name | Returns | | Description/Parameters: |
| getSize | An integer – the size, in number of elements, that the SmartArray currently has the capacity to hold. | | public int getSize()  {  return rgNums.Length;  }  ☺ |
| Note: all methods should be marked **public** | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| **StackOfInts Data & Methods** | | | |
| Data Field Name | Type | | Description: |
| Note: This class must inherit from SmartArray | | | |
| topOfStack | int | | **topOfStack** will be the index of the NEXT space that will be used. So it therefore starts out at 0, meaning that 0 is Unoccupied, but will be the next space used when Push is called.  You're free to make **topOfStack** mean something else, but if you do so, you should clearly document what that meaning is. |
| Note: all data fields should be marked **private** | | | |
| Method Name | | Returns | Description/Parameters: |
| <constructor> | Nothing, by definition | | Allocates the array that the StackOfInts will use to store the integers |
| isEmpty | True, if the stack currently contains NO elements.  False otherwise | | Return type says it all |
| Push | Returns nothing  Will throw an OverflowException if the stack runs out of space in the underlying array | | Parameters:   1. An integer that is the value to be added to the top of the stack   This method will take the value given by the parameter, and add it to the top of the stack.   *Hint:* In order for this to happen, you'll need to adjust **topOfStack** – one of the objectives of this exercise is for you to figure out how. |
| Peek | Returns the top-most item on the stack.  Will throw an UnderflowException if the stack is empty, and therefore there is nothing to Peek at. | | Parameters:   1. None   Note that this method, unlike Pop, does NOT change the stack in any way – it only copies the topmost item into the parameter (if there is a top-most item to copy), and then returns.  *Hint:* Unlike Push or Pop, you will NOT adjust **topOfStack**. |
| Pop | Returns the top-most item on the stack.  Will throw an UnderflowException if the stack is empty, and therefore there is nothing to Pop at. | | Parameters:   1. None   Note that this method, unlike Peek, DOES change the stack –it removes that top-most item from the stack and returns it.  *Hint:* In order for this to happen, you'll need to adjust **topOfStack** – one of the objectives of this exercise is for you to figure out how. |
| Note: all methods should be marked **public** | | | |