

## Assignment 1

In this assignment ~~we~~ you will create a bookstore-application. In the bookstore not only books are sold, but also magazines and CDs.

Create an abstract class “BookStoreItem” and give it the following properties:

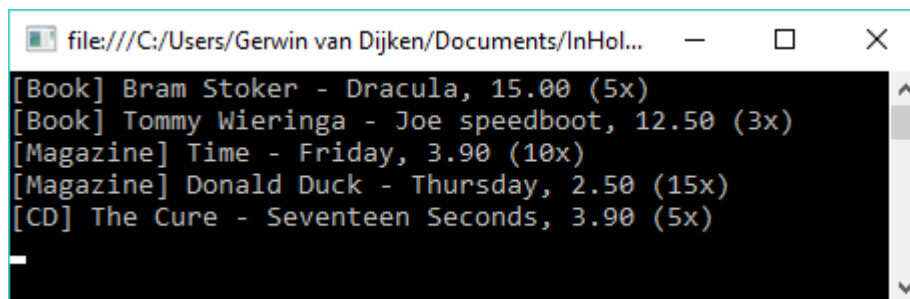
```
public string Title { get; set; }  
public float Price { get; set; }  
public int NumberOfItems { get; set; }
```

This information (title, price, count) is being passed to the constructor of class BookStoreItem.

Create 3 derived classes of BookStoreItem: Book, Magazine and CD. Give class Book an extra (`string`) property “Author”, class Magazine an extra (`DayOfWeek`) property “ReleaseDay”, and class CD an extra property (`string`) “Artist”.

Implement a class “BookStore” that contains a list of “BookStoreItem” objects. Make sure this list can only be accessed by the bookstore class (not accessible by the outside world). Add the next 2 methods to the BookStore class: “Add” (to add an item to the list), and “PrintAllItems” (to print all items).

Test the code by creating a bookstore object (in the Start method), adding several books, magazines and CDs to it, and subsequently print all items. When printing the items, also the item-specific properties must be printed (like the author of a book, and the artist of a CD).



```
file:///C:/Users/Gerwin van Dijken/Documents/InHol...  
[Book] Bram Stoker - Dracula, 15.00 (5x)  
[Book] Tommy Wieringa - Joe speedboot, 12.50 (3x)  
[Magazine] Time - Friday, 3.90 (10x)  
[Magazine] Donald Duck - Thursday, 2.50 (15x)  
[CD] The Cure - Seventeen Seconds, 3.90 (5x)  
_
```

Try also to create an object of class “BookStoreItem”. What kind of error do you see?

## Assignment 2

In this assignment we will implement a Stack. A stack is a data structure in which items can be stored, the last stored item will be the first item to be retrieved from the stack (*and the first stored item will be the last item to be retrieved*). This processing order is known as LIFO (Last-In First-Out).

An example of stack usage is the stack-memory in computer programs; for every method call, the parameters and the local variables (of the method) are stored in the stack. As soon as the method has done its work, these stored items are removed from the stack. Here the same rule applies: items of the last method call are removed first.

We want to be able to perform the following Stack operations (*no matter how the stack is implemented*):

1. `void Push(int value)` → pushes a new item (int) onto the stack; throw an exception when the stack is full
2. `int Pop()` → returns the last pushed item (int) on the stack (and removes this item from the stack); throw an exception if the stack is empty
3. `bool Contains(int value)` → returns true if the stack contains a specific item (int), false otherwise
4. `int Count { get; }` → returns the number of items on the stack
5. `bool IsEmpty { get; }` → returns true if the stack is empty, false otherwise

Create an interface (IStack) with these 5 methods/properties. Create a class (ArrayStack) that implements this IStack interface, using an array for storing the items. Use a constructor parameter for the maximum number of stack items. Use the program below to test your Stack implementation.

```
void Start()
{
    IStack myStack = new ArrayStack(50);
    AddValues(myStack);
    ProcessValues(myStack);
}

void AddValues(IStack stack)
{
    Random rnd = new Random();
    for (int i = 0; i < 10; i++)
    {
        int value = rnd.Next(100);
        stack.Push(value);
        Console.WriteLine($"pushed {value},
                           new count: {stack.Count}");
    }
}

void ProcessValues(IStack stack)
{
    while (!stack.IsEmpty)
    {
        int value = stack.Pop();
        Console.WriteLine($"popped {value}, new count: {stack.Count}");
    }
}
```

```
pushed 76, new count: 1
pushed 4, new count: 2
pushed 39, new count: 3
pushed 38, new count: 4
pushed 0, new count: 5
pushed 87, new count: 6
pushed 30, new count: 7
pushed 68, new count: 8
pushed 28, new count: 9
pushed 33, new count: 10

popped 33, new count: 9
popped 28, new count: 8
popped 68, new count: 7
popped 30, new count: 6
popped 87, new count: 5
popped 0, new count: 4
popped 38, new count: 3
popped 39, new count: 2
popped 4, new count: 1
popped 76, new count: 0
```

Add a test method 'CheckValues' that tests Stack-method 'Contains'.

### Assignment 3

Implement the next interfaces (each in a separate file):

```
public interface IPencil {
    bool CanWrite { get; } // determines if the pencil can still write
    void Write(string message); // writes characters of the message
    void AfterSharpening(); // the pencil is made 'new' (so it can write 'max' again)
}

public interface IPencilSharpener
{
    void Sharpen(IPencil pencil);
}
```

Create a class "Pencil" that implements interface "IPencil", and create a class "PencilSharpener" that implements interface "IPencilSharpener".

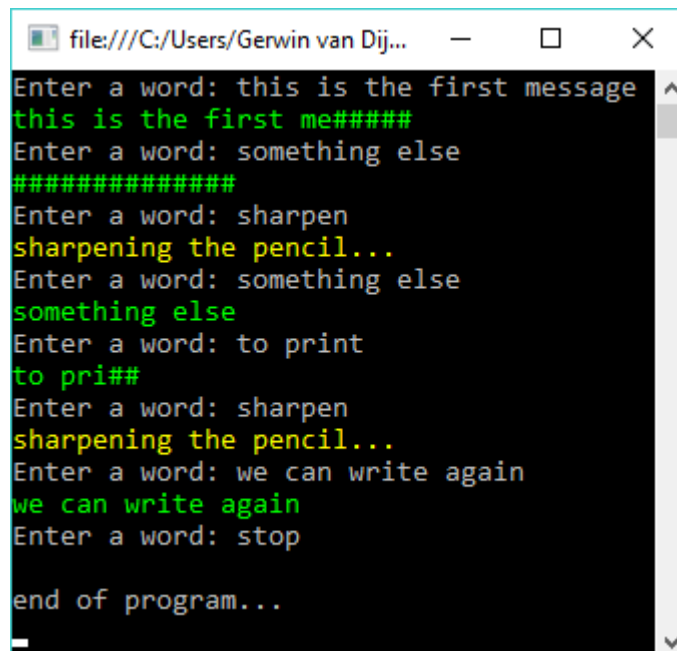
Add the following member fields to class Pencil:

```
private int maxToWrite; // number of characters to write with a sharpened pencil
private int nrOfCharsWritten; // number of written characters
```

The pencil will not be able to write the complete message if it's too long. After writing a certain number of characters, the pencil will become dull and each remaining character will be written as '#'.

Write a small program that reads messages until the user enters the word "stop". Each entered message should be written by the (created) pencil. If the user enters the word "sharpen", then the pencil must be sharpened by the (created) pencil-sharpener.

Output of the program could be like:



```
file:///C:/Users/Gerwin van Dij...
Enter a word: this is the first message
this is the first me#####
Enter a word: something else
#####
Enter a word: sharpen
sharpening the pencil...
Enter a word: something else
something else
Enter a word: to print
to pri##
Enter a word: sharpen
sharpening the pencil...
Enter a word: we can write again
we can write again
Enter a word: stop

end of program...
```