L5-6: Stacks

SESSION	Sessions 5-6
CLASS MODE	PRACTICAL
SLIDES	
Completed	☐
Date	

Stacks

→ Like stack of cards

Stack abstract data type (adt)

- Elements are only added or removed on one end
- Therefore, it is a LIFO data structure
  - Last in - First out
- ADT:
  - Push
  - Pop
  - Peek

Stack implementation

Array

- Fixed size
- Some point of time the array may be full
- Memory
  - Too much allocated (→ wasted)
  - Or not enough (→ maybe resize)

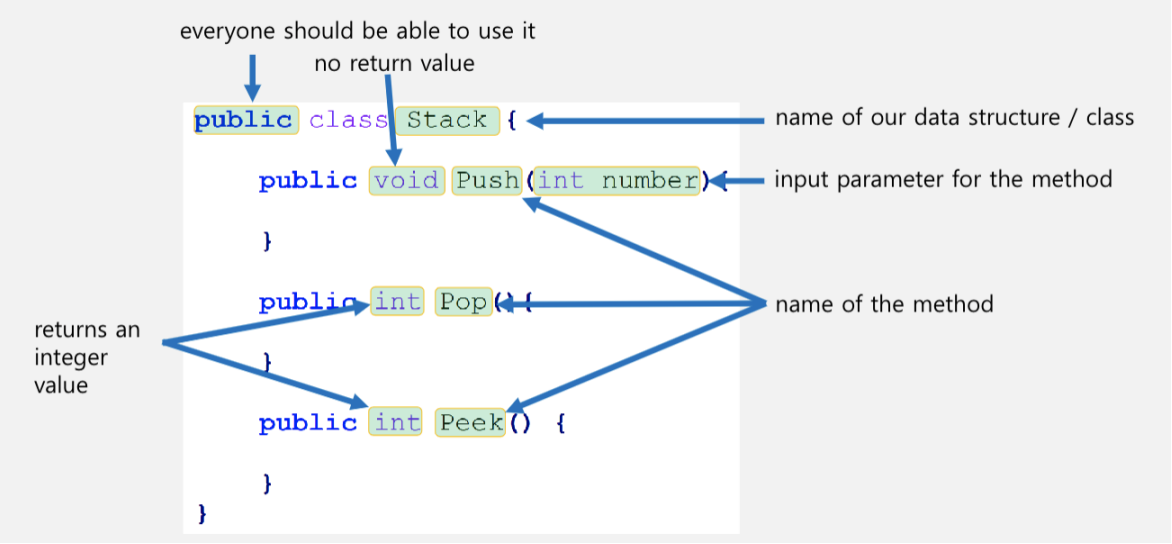
Linked lists

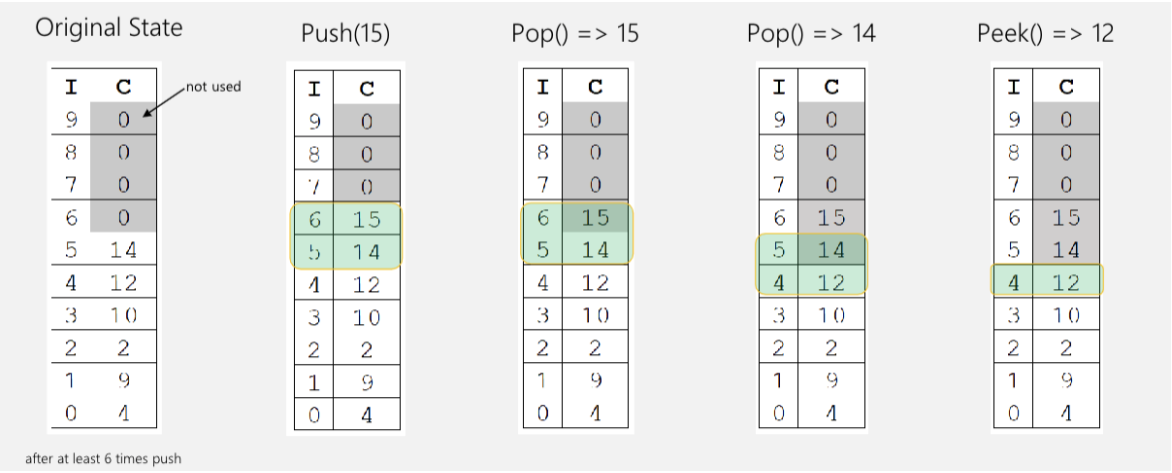
- Dynamic size
- More memory management
- Depending on the size of the concrete data the memory needed for the structure itself could exceed the memory need of data

Creating a Stack class

💡 Notes for Günther's way of making a Stack

▼ Starting point

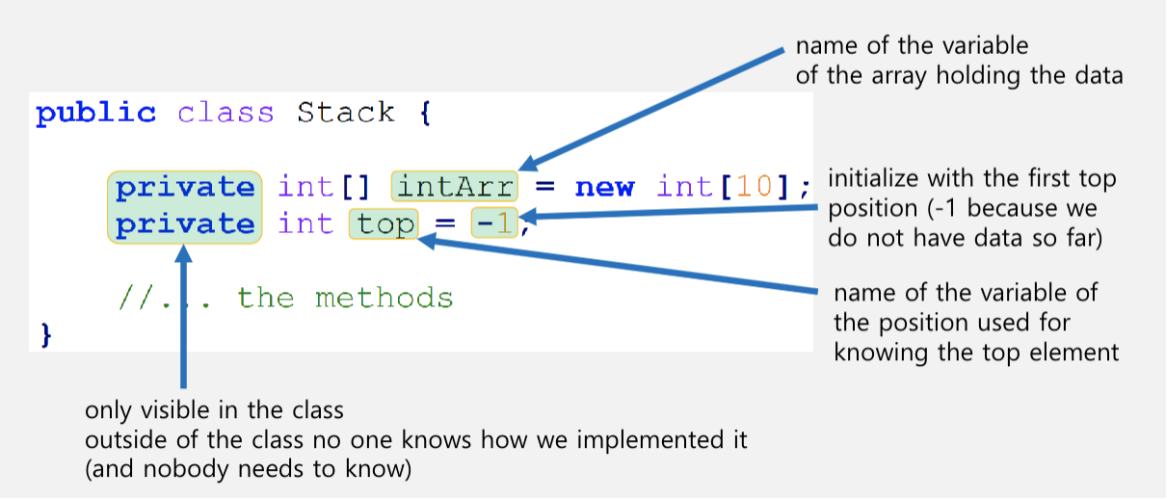




How to create the stack class

1. We need an internal data structure → array
    a. We choose an array because it is our way of storing data, we choose how to manipulate this data in whichever way we want. In this case, we will manipulate it as a Stack (pushing, popping and peeking the top value in the stack.
2. We need at least the variable for the index of the top most element
    a. This is for us to keep track of the index we have the value we want to peek in the stack (the top value)

▼ Visual representation of the things we need



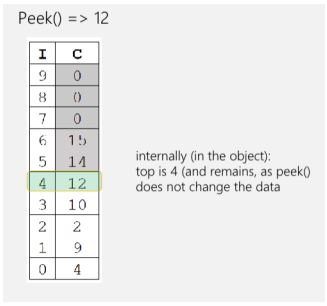
As you can see the initial value of the TopPosition is -1, this is because we need to initialise it, and this number means our stack is empty. As soon as we introduce one value (one integer), this value will go in the first index of our array, index 0, so we will increase the value of the TopPosition to 0.

Peek()

Peek() method: this method returns the top most value. Careful, it doesn't print it, it returns it, this means that if we do $variable = Stack.Peek()$ the value that the function Peek() returns will be stored in variable.

```
public int Peek(){
        int number = StackArray[position];
        return number;
    }
```

▼ **Visual representation of peek**



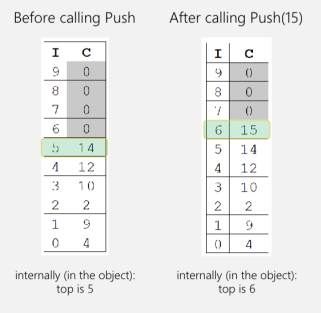
Push()

Push() method: this method adds a new value to the stack, in oder words, appends a new value to the stack. We will do this by increasing the TopPosition by one and assigning that value to the index(TopPosition) of out array.

```
public void Push(int number){
        position = position + 1;
        StackArray[position] = number;
```

```
}
```

▼ **Visual representation of push**

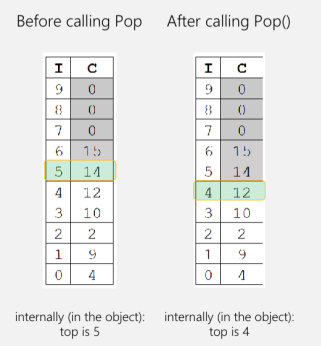


Pop()

💡 **Pop() method:** this method returns the top most value and removes it. As you can see it does the same function as peek, but it decreases TopPosition by one, in this way it is like we removed the value from the stack!
→ Same thing as Peek() we have to be careful because it does not print the value. It returns it. It is important to understand that returning the value means that after the function is executed, the function has a value. For example, if you multiply 2*2, the output is 4, so if you execute Pop() or Peek(), the output will be the top value.

```
public int Pop(){
        int number = StackArray[position];
        StackArray[position] = 0;
        position = position - 1;
        return number;
}
```

▼ **Visual representation of pop**



Refining the Stack Class (Advanced Stack)

💡 Now that we have a first version of the class, we will modify it to make it **more sophisticated**.

Stack Overflow → lifeguard Push() method

💡 **Stack Overflow:** An error condition that occurs when there is no room in the stack for a new item.

💡 We look at this problem whenever we are pushing, and we solve it by checking if the Stack is full, in other words, if the TopPosition is = to the current size of the array. If it is full, in order to push (append a value) we call the DoubleTheInternalArray() method which changes the size of the array and allows us to append.

```
public void Push(int number){
        if (TopPosition == Size-1){
            DoubleTheInternalArray();
        }
        TopPosition ++;
        StackArray[TopPosition] = number;
}
```

Stack Underflow → lifeguard Peek() and Pop() methods.

💡 **Stack underflow:** An error condition that occurs when an item is called for from the stack, but the stack is empty.

💡 We look at this problem whenever we are popping or peeking, and we solve it by checking if the TopPosition is **-1** (this means the Stack is empty), if it is **-1** we do not allow the code to Pop() or Peek()

```
public int Pop(){
        if (TopPosition == -1){
            Console.WriteLine("Could not pop, stack underflow");
            return 0;
        }
        int number = StackArray[TopPosition];
        TopPosition = TopPosition - 1;
        return number;
 }
 public int Peek(){
        if (TopPosition == -1){
            Console.WriteLine("Could not peek, empty stack");
            return 0;
        }
        int number = StackArray[TopPosition];
        return number;
 }
```

Constructors

💡 Another incorporation in the Version2 is the creation of **"CONSTRUCTORS"**. We have two of them, and their job is to initialise the necessary values.
→If you DO NOT input a value whenever you are calling the Class StackV2 from Main, it executes the first Constructor and sets the size of the array to a predetermined value (in this case size=10)
→If you DO pass an integer as an input from Main, this constructor will generate an array with this 'newSize', so we will have a Stack of the wanted size.

```
//StackV2 Constructor: If you DO NOT pass an input this constructor only generates the array for the predetermined 'Size'
    public StackV2()
    {
        Size = 10;
        StackArray = new int[Size];
    }

    //Stack V2() Constructor: If you DO pass an integer as an input, this constructor will generate an array with this 'newSize'
    public StackV2(int newSize)
    {
        Size = newSize;
        StackArray = new int[Size];
    }
```

🌶️ **CAREFUL!** Now you need a variable to store the size of the array (Size) → We declare it at the top of the code.

```
//We declare the internal variables we will use throughout the class
private int Size;
private int[] StackArray;
private int TopPosition = -1;
```

Doubling the size of the array

- The limiting factor for storing more is the size of the array
- We can, internally, resize the array
- But an array is fixed size?

💡 We solve this problem by creating a new array with the size we want, COPYING the old array into the new one (for loop) and then updating the array and the size.

```
//DoubleTheInternalArray() method: This method doubles the size of the 'StackArray' so that we do not overflow the stack
private void DoubleTheInternalArray(){
        int newSize = Size*2;
        int[] newStackArray = new int[newSize];

        for(int i=0; i<=TopPosition; i++){
            newStackArray[i] = StackArray[i];
        }
        StackArray = newStackArray;
        Size = newSize;
        Console.WriteLine("Resized the stack");
        Console.WriteLine("New size; " + newSize);
    }
}
```

🌶️ **CAREFUL!** To copy the array you need to do it element by element, if you do $newStackArray = StackArray$, it will copy the reference not the array.

My codes VisualStudioCode

▼ Version 1 of the Stack class

💡 This is a first implementation of the stack class.
→In Version 1 we only create the main methods in the class and make sure they perform the functions correctly. We have Push(), Pop() and Peek().
→ Then in Version 2 of the stack class we will make it fancy and give it more functionalities. That is because with this implementation we can go on Stack Overflow (going over the allocated size for the array) or Stack Underflow (going under the allocated size for the array). e.g. if the size is 10 and we put index 10 since the array goes from indexes 0-9, the compilation will give us an error.

StackV1

```
internal class Stack
{
    public int[] StackArray;
    private int position = -1;

    public void Push(int number){
        position = position + 1;
        StackArray[position] = number;
```

```
    }

    public int Pop(){
        int numerito = StackArray[position];
        StackArray[position] = 0;
        position = position - 1;
        return numerito;
    }

    public int Peek(){
        int numerito = StackArray[position];
        return numerito;
    }
}
```

Trying the stack class (in main):

```
internal class Program
{
    private static void Main(string[] args)
    {   int[] myarray = new int[10];

        Stack Mystack = new Stack();
        Mystack.StackArray = myarray;
        Mystack.Push(12);

        for(int i=0; i< Mystack.StackArray.Length; i++){
            Console.WriteLine("pos "+ i + ": " + Mystack.StackArray[i]);
        }

    }
}
```

▼ Version 2 of the Stack class (and definitive)

💡 In this version of the stack class we modify the Version 1 so that we fix the Stack Underflow and Overflow problems.
→ **Underflow:** we look at this problem whenever we are popping or peeking, and we solve it by checking if the TopPosition is -1 (this means the Stack is empty), if it is -1 we do not allow the code to Pop() or Peek()
→ **Overflow:** we look at this problem whenever we are pushing, and we solve it by checking if the Stack is full, in other words, if the TopPosition is = to the current size of the array. If it is full, in order to push (append a value) we call the DoubleTheInternalArray() method which changes the size of the array and allows us to append.

🤔 Another incorporation in the Version2 is the creation of “CONSTRUCTORS”. We have two of them, and their job is to initialise the necessary values.
→If you do not input a value whenever you are calling the Class StackV2 from Main, it executes the first Constructor and sets the size of the array to a predetermined value (in this case size=10)
→If you DO pass an integer as an input from Main, this constructor will generate an array with this 'newSize', so we will have a Stack of the wanted size.

StackV2

```
internal class StackV2
{
    private int Size;
    //We declare the internal variables we will use throughout the class
    private int[] StackArray;
    private int TopPosition = -1;

    //StackV2 Constructor: If you DO NOT pass an input this constructor only generates the array for the predetermined 'Size'
    public StackV2()
    {
        Size = 10;
        StackArray = new int[Size];
    }

    //Stack V2() Constructor: If you DO pass an integer as an input, this constructor will generate an array with this 'newSize'
```

```
    public StackV2(int newSize)
    {
        Size = newSize;
        StackArray = new int[Size];
    }

    //Push() method: receives integer 'number', increases variable 'TopPosition' by one and assigns number to that element in the a
    public void Push(int number){
        if (TopPosition == Size-1){
            DoubleTheInternalArray();
        }
        TopPosition ++;
        StackArray[TopPosition] = number;
    }

    //Pop() method: eliminates the top value in the stack, reduces the 'TopPosition' by one and returns
    //the integer that was eliminated
    public int Pop(){
        if (TopPosition == -1){
            Console.WriteLine("Could not pop, stack underflow");
            return 0;
        }
        int numerito = StackArray[TopPosition];
        TopPosition = TopPosition - 1;
        return numerito;
    }

    //Peek() method: returns the top value of our stack
    public int? Peek(){
        if (TopPosition == -1){
            Console.WriteLine("Could not peek, empty stack");
            return null;
        }
        int numerito = StackArray[TopPosition];
        return numerito;
    }

    //PrintStack() method: prints the whole stack
    public void PrintStack(){
        Console.WriteLine("The number of elements in the stack is: " + (TopPosition + 1));
        for(int i =0; i <= TopPosition; i++){
            Console.WriteLine("position " + i + ": " + StackArray[i]);
        }
    }

    //DoubleTheInternalArray() method: This method doubles the size of the 'StackArray' so that we do not overflow the stack
    private void DoubleTheInternalArray(){
        int newSize = Size*2;
        int[] newStackArray = new int[newSize];

        for(int i=0; i<=TopPosition; i++){
            newStackArray[i] = StackArray[i];
        }
        StackArray = newStackArray;
        Size = newSize;
        Console.WriteLine("Resized the stack");
        Console.WriteLine("New size; " + newSize);
    }
}
```

Trying our StackV2

```
internal class Program
{
    private static void Main(string[] args)
    {   Stack Mystack = new Stack();
        Mystack.CreateStack(10);
        Mystack.Pop();


        Mystack.PrintStack();

        Console.WriteLine("--------------Popping");
        Console.WriteLine("pop: " + Mystack.Pop());
        Console.WriteLine("--------------Peeking");
        Console.WriteLine("peek: " + Mystack.Peek());

    }
}
```

▼ Using our stack implementation (exercises)

> Given the array of numbers: {1, 3, 6, 8, 12, 19, 2}
> Write a program (code) which reverses the numbers and write the results to the console.

```
internal class Program
{
    private static void Main(string[] args)
    {
        int[] Guntherstack = new int[]{1, 3, 6, 8, 12, 19, 2};
        int[] ReversedGuntherstack = new int[Guntherstack.Length];

        StackV3 Mystack = new StackV3(Guntherstack.Length);

        for(int i=0; i< Guntherstack.Length; i++){
            Mystack.Push(Guntherstack[i]);
        }
        Console.WriteLine("The reversed array is: ");

        for(int i=0; i< Guntherstack.Length; i++){
            ReversedGuntherstack[i] = Mystack.Pop();
            Console.WriteLine("Position " + i + ":" + ReversedGuntherstack[i]);
        }
    }
}
```

💡 Whenever we have to reverse something, we normally use stacks

▼ Make another Stack Class (for characters: CharStack)

Exercise

> 1. Implement the stack adt for the datatype char, naming it CharStack.
> 2. Write a program, which uses the string: **{ c [s { i (m) } }** and checks if each open parentheses is closed as well. The data type of the parenthesis matters, take the rules you learned in scope.
> **HINTS:** for transforming a string into an array of characters use

```
string input = "{c[s{i(m)}}";
char[] inputAsChar = input.ToCharArray();
```

▼ CharStack implementation

💡 Just had to change some stuff from the StackV2 that we did in class. As you can see, the integer array (int[ ])we had in the StackV2 now is a character array (char[ ]).

```
internal class CharStack
{
     //We declare the internal variables we will use throughout the class
    private int Size = 10;  //Default value for the 'CharStackArray'
    private char[] CharStackArray;
    private int TopPosition = -1;

    //StackV2 Constructor: If you DO NOT pass an input this constructor only generates the array for the predetermined 'Size'
    public CharStack()
    {
        CharStackArray = new char[Size];
    }

    //Stack V2() Constructor: If you DO pass an integer as an input, this constructor will generate an array with this 'newSiz
    public CharStack(int newSize)
    {
        Size = newSize;
        CharStackArray = new char[Size];
    }
```

```
    //Push() method: receives char 'character', increases variable 'TopPosition' by one and assigns 'character' to that elemen
    public void Push(char character){
        if (TopPosition == Size-1){
            DoubleTheInternalArray();
        }
        TopPosition ++;
        CharStackArray[TopPosition] = character;
    }

    //Pop() method: eliminates the top character in the stack, reduces the 'TopPosition' by one and returns
    //the character that was eliminated
    public char Pop(){
        if (TopPosition == -1){
            Console.WriteLine("Could not pop, stack underflow");
            return '0';
        }
        char characterito = CharStackArray[TopPosition];
        TopPosition = TopPosition - 1;
        return characterito;
    }

    //Peek() method: returns the top character of our stack
    public char Peek(){
        if (TopPosition == -1){
            Console.WriteLine("Could not peek, empty stack");
            return '0';
        }
        char characterito = CharStackArray[TopPosition];
        return characterito;
    }

    //PrintStack() method: prints the whole stack
    public void PrintStack(){
        Console.WriteLine("The number of elements in the stack is: " + (TopPosition + 1));
        for(int i =0; i <= TopPosition; i++){
            Console.WriteLine("position " + i + ": " + CharStackArray[i]);
        }
    }

    //DoubleTheInternalArray() method: This method doubles the size of the 'CharStackArray' so that we do not overflow the sta
    private void DoubleTheInternalArray(){
        int newSize = Size*2;
        char[] newCharStackArray = new char[newSize];

        for(int i=0; i<=TopPosition; i++){
            newCharStackArray[i] = CharStackArray[i];
        }
        CharStackArray = newCharStackArray;
        Size = newSize;
        Console.WriteLine("Resized the array");
        Console.WriteLine("New size; " + newSize);
    }
}
```

▼ Main Program Implementation

```
internal class Program
{
    private static void Main(string[] args)
    {
        string GuntherString = "{c[s{i(m)}}";
        char[] GuntherChar = GuntherString.ToCharArray();
        bool Flag = true; //while brackets are opened 'Flag'= True when they are closed 'Flag'=False

        CharStack MyCharStack = new CharStack(GuntherChar.Length);

        for (int i=0; i<GuntherChar.Length; i++){
            if (((GuntherChar[i]=='{') || (GuntherChar[i]=='[') || (GuntherChar[i]=='(')) & Flag==true){
                MyCharStack.Push(GuntherChar[i]);
            }
            else{
                if(((GuntherChar[i]=='}') || (GuntherChar[i]==']') || (GuntherChar[i]==')'))){
                    if ((GuntherChar[i]=='}') & (MyCharStack.Peek()!='{') & (Flag == true)){
                        Console.WriteLine("Bracket " + GuntherChar[i] + " in position " + i + " not in the correct order");
                        Flag = false;
                    }
                    if ((GuntherChar[i]==']') & (MyCharStack.Peek()!='[') & (Flag == true)){
                        Console.WriteLine("Bracket " + GuntherChar[i] + " in position " + i + "not in the correct order");
                        Flag = false;
                    }
                    if ((GuntherChar[i]==')') & (MyCharStack.Peek()!='(') & (Flag == true)){
                        Console.WriteLine("Bracket " + GuntherChar[i] + " in position " + i + "not in the correct order");
                        Flag = false;
                    }
```

```
                    MyCharStack.Pop();
                }
            }
        }

        if (Flag == true){
            Console.WriteLine("All brackets in order");
        }
    }
}
```

▼ **Günther's Main Program Implementation**

```
internal class Program
{
    //Helper function IsOpeningBracket: checks if the character is an opening bracket.
    private static bool IsOpeningBracket(char input){
        char[] brackets = new char[]{'[', '{', '('};

        for (int i=0; i<brackets.Length; i++){
            if (brackets[i]==input){
                return true;
            }
        }

        return false;
    }

    //Helper function IsClosingBracket: checks if the character is a closing bracket.
    private static bool IsClosingBracket(char input){
        char[] brackets = new char[]{']', '}', ')'};

        for (int i=0; i<brackets.Length; i++){
            if (brackets[i]==input){
                return true;
            }
        }

        return false;
    }

    //Helper function MatchesOpeningBracket: checks if two brackets match in type.
    private static bool MatchesOpeningBracket(char opening, char closing){
        if (opening == '[' && closing == ']'){
            return true;
        }
        if (opening == '{' && closing == '}'){
            return true;
        }
        if (opening == '(' && closing == ')'){
            return true;
        }

        return false;
    }

    private static bool IsStringValid(string input) {
            char[] inputAsChars = input.ToCharArray();
            for(int i=0;i<inputAsChars.Length;i++) {
                char charToHandle = inputAsChars[i];
                if(IsOpeningBracket(charToHandle)) {
                    stack.Push(charToHandle);
                }
                else if(IsClosingBracket(charToHandle)) {
                    char topElement = stack.Pop();
                    if(MatchesOpeningBracket(topElement,charToHandle)) {
                        Console.WriteLine(@"found closing " + charToHandle + " and it matches the last opening");
                    }
                    else {
                        Console.WriteLine(@"found closing " + charToHandle + " but the opening " + topElement + " does not mat
                        return false;
                    }
            }
        }
        return true;

    }
    private static void Main(string[] args)
    {

    }
}
```