Workshop Custom Data Structures

Overview

In this workshop we will create our own custom data structures – a custom list (**`SmartArray`**) and a custom stack. The **`SmartArray`** will have similar functionality to **Java `ArrayList`** that you've used before. Our **`SmartArray`** will work only with integers for now, but after the **Generics** lecture from this course, you can try to change that and make the structure generic, which means it will be able to work with any type. It will have the following functionality:

- **`void add(int element)`** - Adds the given element to the end of the list
- **`int get(int index)`** - Returns the element at the specified position in this list
- **`int remove(int index)`** - Removes the element at the given index
- **`bool contains(int element)`** - Checks if the list contains the given element returns **(True or False)**
- **`void add(int firstIndex, int secondIndex)`** - Adds element at the specific index, the element at this index gets shift to the right alongside with any following elements, increasing size
- **`void forEach(Consumer<Integer> consumer)`** - Goes through each one of the elements in the list

Feel free to implement your own functionality or to write the methods by yourself.

The custom stack will also have similar functionality to the Java **`ArrayDeque`** and again, we will make it work only with integers. Later on, we will learn how to implement it in a way that will allow us to work with any types. It will have the following functionality:

- **`void push(int element)`** – Adds the given element to the stack
- **`int pop()`** – Removes the last added element
- **`int peek()`** – Returns the last element in the stack without removing it
- **`void forEach(Consumer<Integer> consumer)`** – Goes through each of the elements in the stack

Feel free to implement your own functionality or to write the methods by yourself.

1. Implement the SmartArray class

Details about the structure

First of all, we must make it clear how our structure should work under the provided public functionality.

- It should hold a **sequence of items in an array**.
- The structure should have **capacity** that **grows twice** when it is filled, **always starting with capacity 4**.

The **`SmartArray`** class should have the fields listed below:

- **`int[] data`** - An array which will hold all of our elements
- **`int size`** – Holds the size with real data of the array
- **`int capacity`** – Holds the size of the array

The structure will have internal methods to make managing of the internal collection easier.

- **`resize`** – this method will be used to increase the internal collection's length twice
- **`shrink`** – this method will help us to decrease the internal collection's length twice
- **`shiftLeft`** – this method will help us to rearrange the internal collection's elements after removing one.
- **`shiftRight`** – this method we will use when we inset element at specific index

Implementation

Create class **SmartArray** and add private static constant field name **INITIAL_CAPACITY** and set the value to **4**. This field is used to declare the **initial capacity** of the **internal** array. It's always **good practice** to use **constants** instead of **magic numbers** in your classes. This approach makes the code better for **managing and reading.**

```
public class SmartArray {
    private static final int INITIAL_CAPACITY = 4;
```

Keep in mind that if the **internal array** has length of **4** this doesn't mean that our collection holds 4 elements. So we need a field which will keep the information of the actual size of the elements in the structure. This field should be updated **every** single time when we **make changes** related to the **count** of the elements like **adding** or **removing**.

```
private int size;
private int capacity;
```

Now let's create the initial **collection,** which is a **private array of type `int`.** To initialize the array, we will use the constructor of the class.

```
private int size;
private int capacity;

public SmartArray() {
    this.data = new int[INITIAL_CAPACITY];
    this.capacity = INITIAL_CAPACITY;
}
```

The whole class with the fields and the constructor should look like that:

```
public class SmartArray {
    private static final int INITIAL_CAPACITY = 4;
    private int[] data;
    private int size;
    private int capacity;

    public SmartArray(){
        this.data = new int[INITIAL_CAPACITY];
        this.capacity = INITIAL_CAPACITY;
    }
}
```

Implement void add(int element) Method

It is time to create the method which **adds** a new elements to the **end** of our collection. It looks like an easy task, but keep in mind that if our internal array is **filled**, we have to **increase it by twice** the length it currently has and **add** the **new element**.

To make our job easier let's create a **`resize()`** method first. The method should be used only **within** the **class** so it must be **private**.

```
private void resize() {

}
```

Here is how the method should work step by step.

- Create a new array with length **twice** the current length of the internal array:

```
private void resize() {
    this.capacity *= 2;
    int[] copy = new int[this.capacity];
}
```

- Iterate through the items in the **internal array** and fill the **newly created array**:

```
for (int i = 0; i < this.data.length; i++) {
    copy[i] = this.data[i];
}
```

- Set the newly created array to the field "**`data`**":

```
this.data = copy;
```

The whole method should look like this:

```
private void resize() {
    this.capacity *= 2;
    int[] copy = new int[this.capacity];

    for (int i = 0; i < this.data.length; i++) {
        copy[i] = this.data[i];
    }

    this.data = copy;
}
```

Now, we are ready to start implementing the logic behind the **`add()`** method. This is how the method should work:

```
public void add(int element) {

}
```

- Check if the **size** of the **actual data** in our **`SmartArray`** is equal to the **`capacity of our collection`**. If it is, this means that the internal array is **filled** and we need to use the **`resize()`** method.

```
if (this.size == this.capacity) {
    this.resize();
}
```

- After we have checked that we have empty space in the internal array, we can just **add** the **new** item at the end and update the "**size**" field:

```
this.data[this.size++] = element;
```

The whole method should look like this:

```
public void add(int element) {
    if (this.size == this.capacity) {
        this.resize();
    }
    this.data[this.size++] = element;
}
```

Before proceeding with the next tasks, it is a good practice, since you've done so much work, to test if everything is fine. Use the **debugger** to **test** for bugs.

Implement int get(index) Method

This method has the functionality to **get item** at given **index** from our internal array. Keep in mind that you have to check if the given index is at the range of our array. If it's in range return the **element** at our index, if not throw the corresponding exception. Lets start implementing our method:

```
public int get(int index) {
    // TODO: Check index

    return this.data[index];
}
```

Checks whether the **index** is **less than zero** or **greater than** the actual **size** of internal array. If its true throw **`IndexOutOfBoundsException`**. We can create **private method `checkIndex`** because we can use it again later:

```
private void checkIndex(int index) {
    if (index < 0 || index >= this.size) {
        // throw IndexOutOfBoundsException
    }
}
```

When we throw any kind of exception we can add our personal message:

```
private void checkIndex(int index) {
    if (index < 0 || index >= this.size) {
        String message = String.format("Index %d out of bounds for length %d",
                index, this.size);
        throw new IndexOutOfBoundsException(message);
    }
}
```

The whole method should look like this:

```
public int get(int index) {
    checkIndex(index);

    return this.data[index];
}
```

Implement int remove(int index) Method

`remove()` method has the functionality to **remove an element** on the **given index** and **returns the same element**. Let's think about how to solve this problem by **dividing it to smaller tasks**.

- First we must check if the index is **valid** and if not, we should throw **`IndexOutOfBoundsException`**
- Get the item on the given index and assign it to a variable, which will be **returned** at the end
- Set the value on the given index to the **default value of `int`**
- Now we have an empty element and we need to **shift** the elements
- Reduce the **size** and check if the size is **4 times smaller** than the **internal array capacity**. If it is we have to think about a way to **shrink** the array
- In the end, return the variable to which we assigned the value of the requested index

So now you already know that we need to implement the other 2 internal methods **`shift()`** and **`shrink()`**.

void shift(int index)

The shift method uses a loop, which moves all of the elements to the left, starting from a given index.

```
private void shift(int index){
    for (int i = index; i < this.size - 1; i++) {
        this.data[i] = this.data[i + 1];
    }
    this.data[size - 1] = 0;
}
```

void shrink()

Decrease the **size** and check if it is **4 times smaller** than the **capacity**. Probably it is but is not necessarily. If its smaller, a good idea is to shrink our array, so we can free some memory. Our **`SmartArray`** will keep only integers, which makes it pretty easy with the memory consumption. However if we had to store complex objects, which would have taken a lot more memory, we would rather think about shrinking it anyway.

The **`shrink()`** method is exactly the same as the **`resize()`** method with the small difference that it will **reduce** the length twice, instead of increasing it:

```
private void shrink() {
    this.capacity /= 2;
    int[] copy = new int[this.capacity];

    for (int i = 0; i < this.size; i++) {
        copy[i] = this.data[i];
    }

    this.data = copy;
}
```

Now we are ready to proceed with the implementation of the **`remove()`** method.

```
public int remove(int index) {

}
```

- First, we need to validate that our index is valid. We can use our **`checkIndex`** method:

```
public int remove(int index) {
    checkIndex(index);


}
```

- Get the value on the given index, assign it to a variable and don't forget to shift the elements:

```
int element = this.data[index];
shift(index);
```

- Now we must reduce the **`size`** and check if **shrinking** the array is **required**:

```
this.size--;

if (this.size <= this.capacity / 4) {
    shrink();
}
```

- After all, just return the element that we saved at the start.

The whole method should look like this:

```
public int remove(int index) {
    checkIndex(index);

    int element = this.data[index];
    shift(index);
    this.size--;

    if (this.size <= this.capacity / 4) {
        shrink();
    }

    return element;
}
```

It is time to test out your **SmartArray** again. If everything works fine, proceed with the next task.

Implement void add(int index, int element) Method

You are already familiar with this method so lets head straight to the implementation. First of all, we will **split** the logic on **small tasks**:

- We have an index parameter, so we must **validate the index**
- We must check if the array should be **resized**
- We have to **rearrange** the items to **free the space for the required index**
- Finally **add** the given element on the index and **increase** the **size**

You probably already noticed, that since we have a method to **rearrange** the elements to the left, used to fill up the empty space when we remove an element, we must have method to **rearrange** elements to the right, so let's create it.

Starting from the **end of the actual elements,** this method will **copy** every single element on the **next indexes**. The loop will **end on the requested index**:

```
private void shiftRight(int index) {
    for (int i = this.size - 1; i > index; i--) {
        this.data[i] = this.data[i - 1];
    }
}
```

Now complete the **add** method:

```
public void add(int index, int element) {
    checkIndex(index);

    if (index == this.size - 1) {
        add(this.data[this.size - 1]);
        this.data[this.size - 2] = element;
    } else {
        this.size++;
        shiftToRight(index);
        this.data[index] = element;
    }
}
```

Implement boolean contains(int element) Method

This method should check **if** the given element **is in the collection**. Return **true** if it is contained and **false** if it's not. It's a simple task, so you should do it all on your own.

Hint: When you are **iterating** through the elements, use the **`size`** field as an **end condition**, instead of the internal array **`capacity`**.

Implement forEach(Consumer<Integer> consumer) Method

Implement a method which will allow you to iterate through each element of the **`SmartArray`**. You can pass **`Consumer<>`** to the method:

```
public void forEach(Consumer<Integer> consumer) {
    for (int i = 0; i < this.size; i++) {
        consumer.accept(this.data[i]);
    }
}
```

Feel free to add methods whatever you may find useful or interesting.

2. Implement the CustomStack class

Details about the structure

The implementation of a **custom stack** is much easier, mostly because you can only execute actions over the last index of the **collection**, plus you can iterate through the collection. You should be able to create it entirely on your own. The first thing you can do is to have a clear vision of how you want your structure to work under the provided public functionality. For example:

- It should hold a **sequence of items in an array**
- The structure should have a **capacity** that **grows twice** when it is filled, **always starting at 4**.

The **`CustomStack`** class should have the fields listed below:

- **`int[] items`** - An array, which will hold all of our elements
- **`int size`** – Holds the **size with real data** of the array
- **`final int initialCapacity`** – This constant's value will be the initial capacity of the internal array

- **`int capacity`** – Holds the size of the array

Implementation

Create a new public class **`CustomStack`** and add a private constant field named **`initialCapacity`** and set the value to **4**. This field is used to declare the **initial capacity** of the **internal** array. We already know that it's not a good practice to have **magic numbers** in your code. Afterwards, we are going to declare our internal array and a field for the **capacity** of elements in our collection:

```
public class CustomStack {
    private static final int INITIAL_CAPACITY = 4;
    private int capacity;
    private int size;
    private int[] items;
}
```

Of course, you have to initialize the collection. Also, set the **capacity** variable to **initial capacity**. As we already know, this can be done inside the constructor of the class:

```
public CustomStack() {
    this.capacity = INITIAL_CAPACITY;
    this.items = new int[this.capacity];
}
```

Next, you have to add a public method **`size`** that holds the value of the **`size`** fields. This way, you will be able to get the size of items in the collection from other classes:

```
public int getSize() {
    return this.size;
}
```

Now you can proceed to the implementation of the methods, which your **`CustomStack`** is going to have. All of the functionalites described in the description are very easy to implement, so we strongly recommend for you to try to do it on your own. If you have any difficulties, you can help yourself with the code snippets below.

Implement void push(int element) Method

This method adds an element to the end of the collection, just like the Java **`ArrayDeque push()`** method does. This is a very easy task. Here is the code you can use, if you meet any difficutlies:

```
public void push(int element) {
    // resize if the size is equal to the capacity
    if (this.size == this.capacity) {
        this.resize();
    }

    this.items[this.size++] = element;
}
```

```
private void resize() {
    this.capacity *= 2;
    int[] copy = new int[this.capacity];

    for (int i = 0; i < this.items.length; i++) {
        copy[i] = this.items[i];
    }

    this.items = copy;
}
```

Implement void pop() Method

The **pop()** method returns the last element form the collection and removes it. The implementation is easier than the implementation of the **remove(int index)** method of the **SmartArray**. Try to implement it on your own. Afterwards, you can look at this implementation:

```
public int pop() {
    // throw NoSuchElementException if the size is zero

    int element = this.items[this.size - 1];
    this.size--;
    return element;
}
```

```
private void checkEmpty() {
    if (this.size == 0) {
        throw new NoSuchElementException("CustomStack is empty");
    }
}
```

Implement void peek() Method

The **peek()** method has the same functionality as the **Java ArrayDeque** - it returns the last element from the collection, but it **doesn't remove** it. The only thing we need to consider is that you can't get an element from an empty collection, so you must make sure you have the proper **validation**. For sure, you will be able to implement it on your own.

Implement void forEach(Consumer<Integer> consumer) Method

This method goes through every element from the collection and executes the given action. The implementation is very easy, but it requires some additional knowledge, so here is what you can do:

```
public void forEach(Consumer<Integer> consumer) {
    for (int i = 0; i < this.size; i++) {
        consumer.accept(this.items[i]);
    }
}
```

You can add any kind of functionalities to your **`CustomStack`** and afterwards you can test how it works in your **`main()`** method.

"Imagination will often carry us to worlds that never were, but without it we go nowhere."

Carl Sagan