Exceptions and Design

BIU OOP 2018
Table of contents

1 Error Handling
   ■ Overview
   ■ Exceptions
   ■ RuntimeExceptions

2 Design
   ■ Command query separation
   ■ Separation of concerns
Exceptions
Why did we implement a **Positive** Integer stack?
Stack limitation

- Why did we implement a **Positive** Integer stack?
- We had to use the value ’-1’ for **error handling**
Why did we implement a **Positive** Integer stack?

- We had to use the value `-1` for **error handling**
- Shouldn’t there be a **better** way to do error handling?
What is an Exception?

- A better way to **announce** "some error" happened
  - Better than using special **return** values (i.e. return -1;)
- Gives the ability to **respond** to errors only where it’s appropriate
- Separate normal code execution from error handling
Announcing that an error happened

- The correct terminology is **throwing an Exception**

- An **Exception** is an Object containing information about what happened (and where)

```
return -1;
```

**VS**

```
throw new Exception("Something happened");
```
Part of the method signature (just like the return type)

```java
int factorial(int n) throws Exception {
    if (n < 0) {
        throw new Exception("Negative number is not allowed");
    } else if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

(Show compilation error when we forget the declaration)
Exception method signature

- Same way as a return type is part of the method signature
- If you don't handle it, you must declare you may throw an Exception

```java
int factorial(int n) throws Exception {
    if (n < 0) {
        throw new Exception("Negative number is not allowed");
    } else if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
void printFactorial(int n) throws Exception {
    int result = factorial(n);
    System.out.println(result);
}
```
Responding to errors

- The correct terminology is **catching an Exception**
- Achieved by using a try/catch block

```java
int factorial(int n) throws Exception {
    if (n < 0) {
        throw new Exception("Negative number is not allowed");
    } else if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

void printFactorial(int n) {
    try {
        int result = factorial(n);
        System.out.println(result);
    } catch (Exception e) {
        System.out.println("Error");
    }
}
```
void printFactorial(int n) {
    try {
        int result = factorial(n);
        System.out.println(result);
    } catch (Exception e) {
        System.out.println("Error");
    }
}

Note
When an exception will be thrown from factorial, no other lines of code inside the try block will be executed, the program jumps straight to the catch block.
Stack using Exceptions

- Lets improve the Stack we made last time.
- We can remove the **Positive** from `BoundedPositiveIntegerStack`.
New implementation of stack push

```java
/**
 * push an element into the stack.
 *
 * @param element the element to push into the stack.
 * @throws Exception if no space is left in stack.
 */
public void push(int element) throws Exception {
    // check if we have space to add a new element
    if (numberOfElements == this.elements.length) {
        throw new Exception("No space left in stack");
    }

    // add the new element
    this.elements[this.numberOfElements] = element;
    this.numberOfElements += 1;
}
```
New implementation of top

```java
/**
 * The top element in the stack.
 * @return the element on top of stack.
 * @throws Exception if stack is empty.
 */
public int top() throws Exception {
    // check if we have any elements in the stack
    if (this.numberOfElements == 0) {
        // return an indication that the stack is empty
        throw new Exception("Stack is empty");
    }
    // return the correct element
    return this.elements[this.numberOfElements - 1];
}
```
### New implementation of pop

```java
/**
 * Pop element from the stack.
 * @return the element on top of stack.
 * @throws Exception if stack is empty.
 */
public int pop() throws Exception {
    // get the element on top of the stack
    int top = top();
    // decrement the number of elements, unless the stack is already empty
    this.numberOfElements -= 1;
    // return the element
    return top;
}
```

**Note**

If `top()` throws Exception, `numberOfElements` value will not be affected, the code after `top()` will not be executed and we will immediately exit the method.
Let’s look at the complete code - `exceptions.BoundedIntegerStack.java`

Let’s look at the how it can be used - `exceptions.StackUsageExample.java`
Stack using Exceptions

- Let’s look at the complete code - `exceptions.BoundedIntegerStack.java`
- Let’s look at how it can be used - `exceptions.StackUsageExample.java`

**Note**

Would we want to use such an implementation? So much unnecessary code! There must be a better way!
Exception vs RuntimeException

- **Exception**
  - Must be handled (try/catch)
  - Must be declared on methods (throws)

- **RuntimeException**
  - Don’t have to be handled (will still crash our code)
  - Don’t have to be declared (but probably should)
Let’s look at the complete code -
runtimeexceptions.BoundedIntegerStack.java

Let’s look at the how it can be used -
runtimeexceptions.StackUsageExample.java

Note
No unnecessary code, if application crashes we will have the exact reason!
Why not always use RuntimeExceptions?

- Some actions are **unpredictable by definition** (for example opening a file)
- We want to force the user of our class to handle the problems and not ignore them.
Code crashes

- In Java no such thing as a code crash
- Everything causes a `RuntimeException`
- (Show Examples - ExceptionExamples.java)
Bad practices

- Never use the try/catch as a control flow
- An empty catch clause is a bad idea, you’re just ignoring the problem

```java
try {
    while (true) {
        System.out.println(stack.pop());
    }
} catch (RuntimeException e) {
    // Do nothing, the stack is just empty
}
```
Bad practices

- Never use the try/catch as a control flow
- An empty catch clause is a bad idea, you’re just ignoring the problem

```java
1 try {
2   while (true){
3     System.out.println(stack.pop());
4   }
5 } catch (RuntimeException e){
6    // Do nothing, the stack is just empty
7 }
```

- A better approach

```java
1 while (!stack.isEmpty()){  
2   System.out.println(stack.pop());
3 }
```
Design

Command Query Separation
Command Query Separation

Separate the methods of a class into two categories:

- **Query**: Ask a question about the state of an object. Should not alter the state of the object.
- **Command**: Change the state of the object in some way. Should not return any value.

**Why?**
- Better design
- Easier to test
Command Query Separation

Separate the methods of a class into two categories:

- **Query**
  - Ask a question about the state of an object
  - Should not alter the state of the object
Command Query Separation

Separate the methods of a class into two categories:

- **Query**
  - Ask a question about the state of an object
  - Should not alter the state of the object

- **Command**
  - Change the state of the object in some way
  - Should not return any value
Command Query Separation

Separate the methods of a class into two categories:

- **Query**
  - Ask a question about the state of an object
  - Should not alter the state of the object

- **Command**
  - Change the state of the object in some way
  - Should not return any value

- **Why?**
  - Better design
  - Easier to test
Command Query Separation - Stack

- **Stack Queries**
  - top()
  - size()
  - isEmpty() - *derived query* (uses a basic size query)

Note: Can we do something to improve the design?
Command Query Separation - Stack

- **Stack Queries**
  - `top()`
  - `size()`
  - `isEmpty()` - **derived query** (uses a basic size query)

- **Stack Commands**
  - `push()`
  - `pop()` - is it a command if it returns a value?
Command Query Separation - Stack

- **Stack Queries**
  - top()
  - size()
  - isEmpty() - **derived query** (uses a basic size query)

- **Stack Commands**
  - push()
  - pop() - is it a command if it returns a value?

**Note**
Can we do something to improve the design?
Command Query Separation - Stack

- **Stack Queries**
  - top()
  - size()
  - isEmpty() - **derived query** (uses a basic size query)

- **Stack Commands**
  - push()
  - remove()

- **Derived methods**
  - pop() - uses top() query and a remove() command
Improved Stack implementation

- New method **remove**

  ```java
  public void remove() throws RuntimeException {
      // check if we have any elements in the stack
      if (this.numberOfElements == 0) {
          throw new RuntimeException("Stack is empty");
      }
  }
  ```

- New implementation of **pop**

  ```java
  public int pop() throws RuntimeException {
      // get the element on top of the stack
      int top = top();
  }
  ```
Unit Tests

- We should have an automatic test for our Stack
- See `querycommand.BoundedIntegerStackTest`
Design
Separation of concerns
Unbounded Stack

- Let's make the use of our stack even simpler!
- Why limit ourselves to deciding the maximum size at the beginning?
- What can we do?
Unbounded Stack

- Let's make the use of our stack even simpler!
- Why limit ourselves to deciding the maximum size at the beginning?
- **What can we do?**
  - Add a `resizeTo(int newSize)` method?
Unbounded Stack

- Lets make the use of our stack even simpler!
- Why limit ourselves to deciding the maximum size at the beginning?
- **What can we do?**
  - Add a `resizeTo(int newSize)` method?
  - Why not make it **transparent** for our user?
**Unbounded Stack**

- **New strategy**
  - Start with a small default size

```java
22     IntegerStack() {
23         this.elements = new int[DEFAULT_NEW_STACK_SIZE];
24         // initialize number of elements to 0 (empty stack)
25         this.numberOfElements = 0;
26     }
```

**Note**

We will completely remove the second Constructor

`IntegerStack(int maxSize)`
Unbounded Stack - signatures

```java
22   IntegerStack() {
33   public void push(int element) {
63   public int pop() throws RuntimeException {
80   public int top() throws RuntimeException {
96   public void remove() throws RuntimeException {
111  public int size() {
```
### Unbounded Stack

- **New strategy**
  - Every time we run out of space, let's increase the array by a factor of 2

```java
public void push(int element) {
    // do we have enough space for a new element?
    if (numberOfElements == this.elements.length) {
        // allocate a bigger array to hold our elements (twice the size)
        int newSize = this.elements.length * 2;
        int[] tmp = new int[newSize];

        // copy old values to our new array
        for (int i = 0; i < this.elements.length; i += 1) {
            tmp[i] = this.elements[i];
        }

        // replace reference, old array will be garbage collected
        this.elements = tmp;
    }

    // add the new element
    this.elements[this.numberOfElements] = element;
    this.numberOfElements += 1;
}
```
Separation of Concerns

- A design principle that says that each module should address a separate concern
- Improves code re-usability
- Better code maintainability
- Code that is easier to understand
What are the concerns in our latest *IntegerStack* implementation?
What are the concerns in our latest *IntegerStack* implementation?

- Stack functionality concern
What are the concerns in our latest IntegerStack implementation?

- Stack functionality concern
- Array resizing concern
What should we do?

Create a new class `UnboundedArray` with methods `get(int index)` and `set(int index, in value)`.

Use the new `UnboundedArray` in our `IntegerStack` implementation.

We can now use our `UnboundedArray` in other places.

Show Examples:
- `separationofconcerns.UnboundedIntegerArray`
- `separationofconcerns.IntegerStack`
What should we do?

- Create a new class *UnboundedArray*
  - get(int index)
  - set(int index, in value)
Separation of Concerns - Unbounded Stack 2

What should we do?

- Create a new class UnboundedArray
  - get(int index)
  - set(int index, in value)
- Use the new UnboundedArray in our IntegerStack implementation
- We can now use our UnboundedArray in other places
What should we do?
- Create a new class `UnboundedArray`
  - `get(int index)`
  - `set(int index, in value)`
- Use the new `UnboundedArray` in our `IntegerStack` implementation
- We can now use our `UnboundedArray` in other places

Show Examples:
`separationofconcerns.UnboundedIntegerArray`
`separationofconcerns.IntegerStack`
Test changes - again

We already have a Test, let’s run it to see we didn’t break anything!
Summary

1 Error Handling
   - Overview
   - Exceptions
   - RuntimeExceptions

2 Design
   - Command query separation
   - Separation of concerns