

CS1020: DATA STRUCTURES AND ALGORITHMS I

Tutorial 6 – Exception, Linked List Variants

(Week 8, starting 7 March 2016)

1. Exceptions

Examine the following code, especially how `calculate()` works:

```
class DivideBy4Exception extends Exception {}
class DivideBy100Exception extends DivideBy4Exception {}
class DivideBy400Exception extends DivideBy100Exception {}

/* Part (e): Poor design! DO NOT EVER Use Exceptions this way!! */

public class Mystery {

    public int calculate(int input) {
        int value = 0;
        for (int index = 1; index <= input; index++)
            value += buzz(index);
        return value;
    }

    public int buzz(int value) {
        try {
            if (value % 400 == 0)
                throw new DivideBy400Exception();
            if (value % 100 == 0)
                throw new DivideBy100Exception();
            if (value % 4 == 0)
                throw new DivideBy4Exception();
            return 0;
        } catch (DivideBy4Exception ex) {
            return 1;
        } catch (DivideBy100Exception ex) {
            return 0;
        } catch (DivideBy400Exception ex) {
            return 1;
        } finally {
            return 3;
        }
    }

    public static void main(String[] arr) { ... }
}
```

- (a) Why can't the program be compiled? Identify and **fix the error**, **THEN** attempt (b) to (e).
- (b) What do `calculate()` and `buzz()` do? What is the result of `calculate(2000)`, why?
- (c) **IF the finally clause is removed**, what will the result of `calculate(2000)` be, why?
- (d) If we do not want to handle Exceptions within `buzz()`, what alternatives are there?
- (e) Why is this program poorly designed? Why should we **NOT** write a program like this?

Answer

Related Concepts

- Catch specific Exceptions first
- Try-catch-finally flow
- Checked exceptions, **throws** keyword

(a)

With this code, the most general Exception (highest in the inheritance hierarchy) would be caught first. The other 2 catch blocks have no chance to catch the more specific exceptions. We need to **reverse the order** of the 3 catch blocks. That way, the **specific Exceptions will be first caught and handled specifically**, while the most general exceptions are checked for last.

(b)

`calculate(...)` sums up `buzz(1)` to `buzz(input)`, in an attempt to calculate the number of leap years in `[1, input]`. `buzz(...)` always returns 3, because the finally block is executed even when a return statement is executed within the try or catch blocks. Therefore, the result of `calculate(2000)` is 6,000.

The **finally block is “always” executed, EVEN** when **return, continue, or break** are executed within the try or catch blocks. The few cases where the finally block are not executed are:

- A call to `System.exit(...)`;
- Infinite loop
- Java (JVM) itself throws an error, or crashes

In this case, `buzz()` will **return the return value in the finally block**, not that of the try or catch block.

(c) It will be 485, the number of leap years in `[1,2000]`. $485 = 2000 / 4 - 2000/100 + 2000/400$.

(d)

`buzz()` can allow the Exception to be **thrown to the caller**. As the 3 Exceptions are all not `RuntimeException`, they are **Checked Exceptions**. The compiler checks to ensure that these Exceptions are handled somewhere. Therefore, `buzz()` should **indicate throws** `DivideBy4Exception`, so that the caller is aware of the most general Exception that needs to be handled.

`calculate()` then has the responsibility to handle the Exceptions. If it does not do so, it can likewise indicate **throws** `DivideBy4Exception` and allow the caller to handle the Exception (but the output would now be “incorrect”).

(e)

Exceptions should not be a substitute for proper control flow logic, such as if/else, switch-case, break and continue statements.

2. List ADT Implementations

In lectures, we have learned two List implementations – array-based and reference-based. ArrayList and Vector are array-based list implementations, while LinkedList is a reference-based implementation. Let us compare and contrast both implementations.

For a list containing N elements, around how many elements would be accessed/modified when:

- (a) Adding to end of the List (new index == N / tail)
- (b) Adding to front of the List (index == 0 / head)
- (c) Removing from front of the List (index == 0 / head)
- (d) Getting (accessing) any element, on average (from index == 0 to index == $N-1$)

Answer

We are assuming that the linked list as a tail reference.

- (a) For array-based implementation, **most of the time** only 1 element is accessed/modified. However, when the capacity of the array is used up to its full, the entire underlying array has to be **reallocated** to another array with a larger capacity, which involves copying all N elements to the new array.

When adding many elements, **on average** the number of elements modified is **still a constant**.

For a linked list, only 1 node is accessed, or 2 nodes (if you consider the new node to be also “accessed”).

- (b) In a typical array-based implementation, N elements have to be copied away from the front (to the right) before insertion. For a linked list, only **1 or 2** nodes are accessed/modified.

- (c) In an **array-based** implementation, (just like in (b)) $N-1$ elements have to be copied towards the front (to the left). For a **linked list**, only **1 or 2** nodes are accessed/modified.

- (d) In an **array-based** implementation, only **1** access is needed to move to any index.

For a linked list, the average number of accesses is $N/2$ using the correct iterators.

For a doubly linked list (with head and tail), the average number of accesses is $N/4$.

Therefore, insertion/removal from one end of an array-based List is inefficient. On the other hand, array-based Lists are good for random accesses.

3. Circular Linked List

Implement a method `swap(int index)` in the `CircularLinkedList<E>` class given to you, to swap the node at the given index with the next node.

If the index is larger than the size of the list, then the index wraps around. For example, if the list has 13 elements, then `swap(15)` will swap nodes at indexes 2 and 3.

```
public class CircularLinkedList<E> {

    public int _size;
    public ListNode<E> _head;
    public ListNode<E> _tail;

    public void addFirst(E element) {
        _size++;
        _head = new ListNode<E>(element, _head);
        if (_tail == null)
            _tail = _head;
        _tail.setNext(_head);
    }

    public String toString() { ... }
    public void swap(int index) { ... }
}
```

A pre-condition is that index will be non-negative ($\text{index} \geq 0$). **Restriction:** You are **NOT allowed** to create any **new nodes**, and you are NOT allowed to **modify the element** in any node.

[Hint: Consider all cases, and remember to update the necessary instance attributes!]

Answer

```
public void swap(int index) {
    if (_size < 2)
        return; // if 0 or 1 nodes, don't bother

    if (_size == 2) {
        ListNode<E> newTail = _head;
        _head = _tail;
        _tail = newTail;
        return;
    } // if 2 nodes, only need to swap head and tail references

    // At this point, the list has >2 nodes
    index %= _size; // ensure that index < size of list first

    // get the 3 desired nodes
    ListNode<E> prev = _tail;
    for (int loopIdx = 0; loopIdx < index; loopIdx++)
        prev = prev.getNext();
    ListNode<E> curr = prev.getNext(); // curr now at indexed node
    ListNode<E> succ = curr.getNext();

    // swap the 2 nodes: Note the order!
    curr.setNext(succ.getNext());
    succ.setNext(curr);
    prev.setNext(succ);

    if (index == 0) {
        _head = succ; // head incorrect
    } else if (index == _size - 2) {
        _tail = curr; // swap(tail-1), tail incorrect
    } else if (index == _size - 1) {
        _head = curr; // swap(tail), both head & tail swapped
        _tail = succ;
    }
}
```

