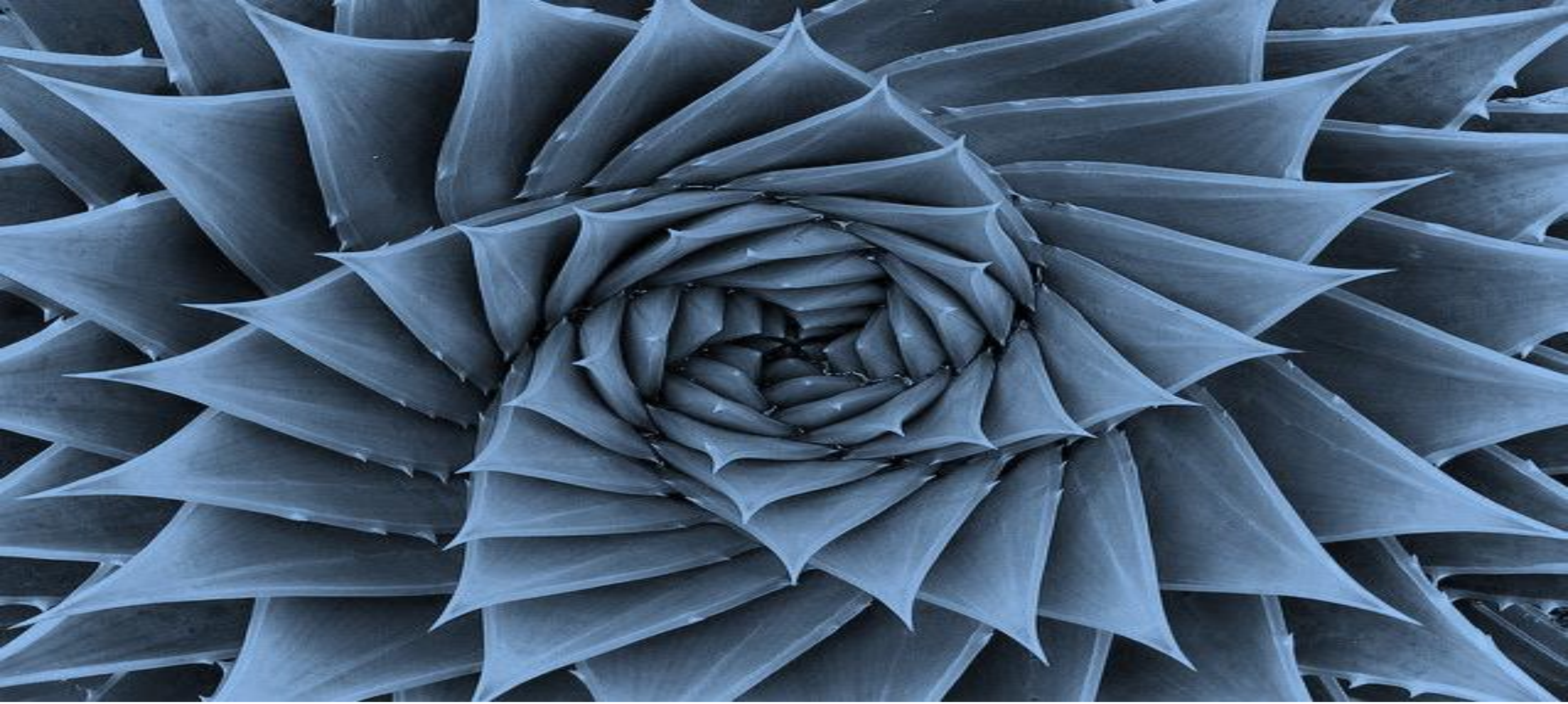


Advanced Object-Oriented Programming

CPT204 – Lab 3
Erick Purwanto

Xi'an Jiaotong-Liverpool University
西交利物浦大学

CPT204 Advanced Object-Oriented Programming

Lab 3

Recursion

Welcome !

- Welcome to Lab 3 !
    - We are going continue practising with testing while reviewing recursion and recursion + helper method

- You will find in this lab
    1. Lab Exercise 3.1, 3.2 and their hints
    2. Exercise 3.1 - 3.5

- Download **lab3** zip file from Learning Mall
- Don't forget to import the **lab3** files and the library into an IntelliJ project
    - Read **lab1** again for reference

Lab Exercise 3.1 CheckSubstring

- Given an input string, a ***non-empty*** substring subs, and a non-negative number ***n***, compute *recursively* and return **true** if and only if at least ***n*** copies of subs occur in the string, possibly with ***overlapping.***

  Do **not** use loops (do not write **for** or **while** anywhere in your code).
  Do **not** use any regular expressions and methods such as matches, split, replaceAll.

- Test case 1:
  checkSubstring("abcxxxxabc", "abc", 2) → true

- Test case 2:
  checkSubstring("abcxxxxabc", "abc", 3) → false

- Test case 3:
  checkSubstring("ababaxxxaba", "aba", 3) → true

Lab Exercise 3.1 CheckSubstring

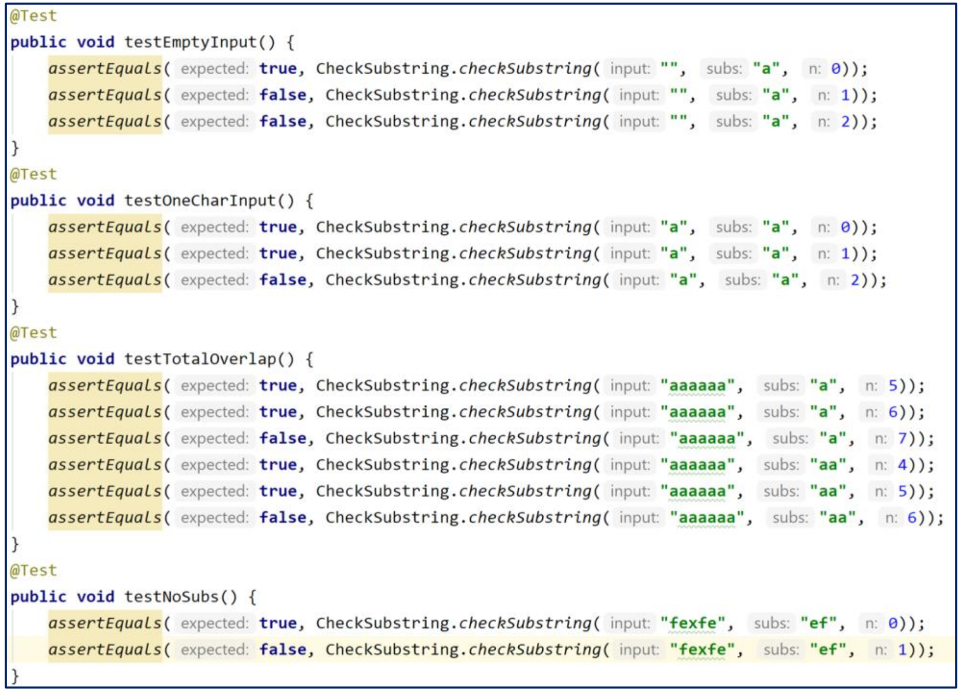
- Skeleton code:

```
/**
 * Check if at least n copies of a substring occur in input string.
 * They may overlap.
 * For example, checkSubstring("ababaxxxaba", "aba", 3) → true
 * @param input is the input string.
 * @param subs is the non-empty substring.
 * @param n is non-negative number of occurrences.
 * @return true iff at least n copies of subs occur in input.
 */
public static boolean checkSubstring(String input, String subs, int n) {
    // base case

    // recursive step
}
```

Continue with Test-Driven Programming

We use the same approach as last week, write the test code first

- Open CheckSubstringTest.java, create and add more test cases, for example:



```
@Test
public void testEmptyInput() {
    assertEquals( expected: true, CheckSubstring.checkSubstring( input: "", subs: "a", n: 0));
    assertEquals( expected: false, CheckSubstring.checkSubstring( input: "", subs: "a", n: 1));
    assertEquals( expected: false, CheckSubstring.checkSubstring( input: "", subs: "a", n: 2));
}
@Test
public void testOneCharInput() {
    assertEquals( expected: true, CheckSubstring.checkSubstring( input: "a", subs: "a", n: 0));
    assertEquals( expected: true, CheckSubstring.checkSubstring( input: "a", subs: "a", n: 1));
    assertEquals( expected: false, CheckSubstring.checkSubstring( input: "a", subs: "a", n: 2));
}
@Test
public void testTotalOverlap() {
    assertEquals( expected: true, CheckSubstring.checkSubstring( input: "aaaaaa", subs: "a", n: 5));
    assertEquals( expected: true, CheckSubstring.checkSubstring( input: "aaaaaa", subs: "a", n: 6));
    assertEquals( expected: false, CheckSubstring.checkSubstring( input: "aaaaaa", subs: "a", n: 7));
    assertEquals( expected: true, CheckSubstring.checkSubstring( input: "aaaaaa", subs: "aa", n: 4));
    assertEquals( expected: true, CheckSubstring.checkSubstring( input: "aaaaaa", subs: "aa", n: 5));
    assertEquals( expected: false, CheckSubstring.checkSubstring( input: "aaaaaa", subs: "aa", n: 6));
}
@Test
public void testNoSubs() {
    assertEquals( expected: true, CheckSubstring.checkSubstring( input: "fexfe", subs: "ef", n: 0));
    assertEquals( expected: false, CheckSubstring.checkSubstring( input: "fexfe", subs: "ef", n: 1));
}
```

WARNING: Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

Lab Exercise 3.1 CheckSubstring Hints

- The idea is to check whether subs is the prefix of input string, recursively call the method always on input with one less letter; and on one less n if subs is the prefix
- Base case, therefore, is when length of input is less than subs
  Depending on n, if n is positive, which means not enough subs found, return false; otherwise, n zero or negative, found enough or even more, return true
- Recursive step, two cases:
  - subs is a prefix, return recursive call on one less n,
  - subs is not prefix, return recursive call on original n,

  both calling on the input string shifted by 1, since we count overlaps

Lab Exercise 3.2 EqualSum

- Given a list of integers, you want to know whether it is possible to divide the integers into two sets, so that the sums of the two sets are the same.
  Every integer **must be in** one set or the other.
  Write a *recursive* helper method that takes ***any number of arguments you like***, and make the initial call to your recursive helper method from equalSum().
  Do **not** use any loops or regular expressions.

- Test case 1:
  equalSum([2, 3, 5]) → true

- Test case 2:
  equalSum([2, 2, 5]) → false

Lab Exercise 3.2 EqualSum

- Given a list of integers, you want to know whether it is possible to divide the integers into two sets, so that the sums of the two sets are the same. Every integer **must be in** one set or the other. Write a *recursive* helper method that takes ***any number of arguments you like***, and make the initial call to your recursive helper method from equalSum(). Do **not** use any loops or regular expressions.
- Test case 1:
  equalSum([2, 3, 5]) → true

  since {2, 3} and {5} have an equal sum
- Test case 2:
  equalSum([2, 2, 5]) → false

  the sets cannot be {2} and {2} 5 must be in a set

Lab Exercise 3.2 EqualSum

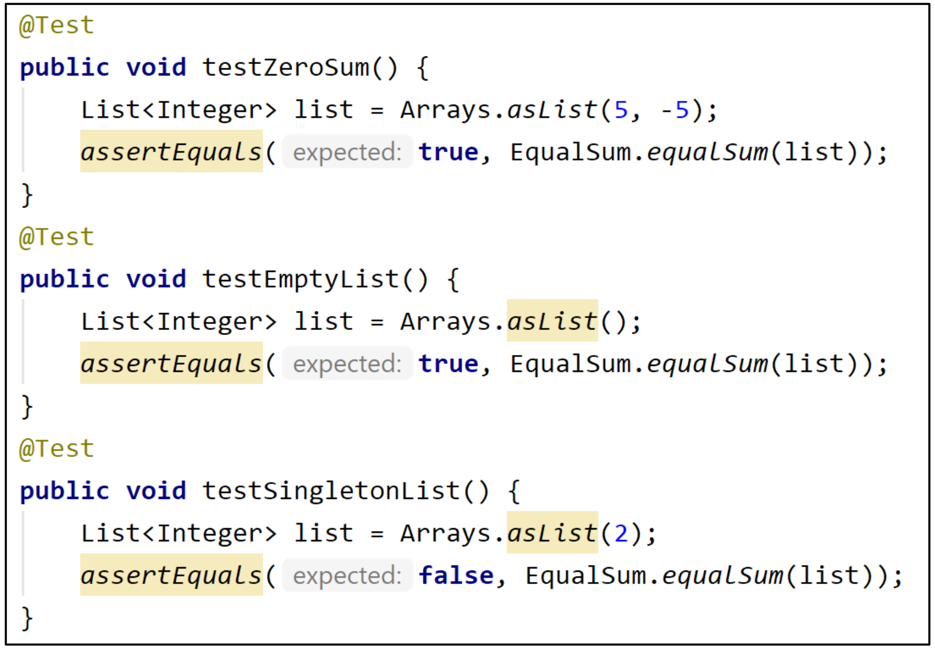
- Skeleton code:

```java
/**
 * Decide if it is possible to divide the integers in a list into two sets,
 * so that the sums of the two sets are the same.
 * Every integer must be in one set or the other.
 * For example, equalSum([2, 3, 5])  →  true.
 * @param list is a list of integers.
 * @return true iff there are two sets having the same sum.
 */
public static boolean equalSum(List<Integer> list) {

    // call your recursive helper method


}
```

Continue with Test-Driven Programming

We use the same approach as last week, write the test code first

- Open EqualSumTest.java, create and add more test cases, for example:



```
@Test
public void testZeroSum() {
    List<Integer> list = Arrays.asList(5, -5);
    assertEquals( expected: true, EqualSum.equalSum(list));
}
@Test
public void testEmptyList() {
    List<Integer> list = Arrays.asList();
    assertEquals( expected: true, EqualSum.equalSum(list));
}
@Test
public void testSingletonList() {
    List<Integer> list = Arrays.asList(2);
    assertEquals( expected: false, EqualSum.equalSum(list));
}
```

WARNING: Hints to the exercise on the next slide

Please try to solve the exercise by yourself first...

Lab Exercise 3.2 EqualSum Hints

- One solution is to use a recursive helper method with 4 parameters: we pass the list, the usual starting index, and two sums: sum1 and sum2.
  - The original method just call it with initial start and sums equal zero

- Since we're going to increment start, the base case is when start exceeds the valid index, in which case we return true iff the two sums are equal.

- Two cases for the recursive step:
  - we add element at index start to the first sum,
  - or the second sum,

    and just pass the other sum unmodified.

  So we do two recursive calls to list starting start+1, and return true iff at least one of those two cases are true.

Week 3 Online Programming Exercises

- Start with creating a good set of test cases first !
    - Include the corner cases, such as empty list or empty string ;
      and corner cases for the elements of the list ;
      as well as a test case for every possible subdomain

- Use IntelliJ for testing

- Do **not** write <u>for</u> or <u>while</u> anywhere in your code

Exercise 3.1 CountBabaMama

- Given a string, write a method that returns the number of occurrences of substrings "baba" or "mama" in the input string **recursively**.
  They **may** overlap.
  Do **not** use any loops within your code.
  Do **not** use any regular expressions and methods such as matches, split, replaceAll.

- Test case 1:
  countBabaMama("aba babaa amama ma") → 2

- Test case 2:
  countBabaMama("bababamamama") → 4

  (2 babas that overlap, plus 2 mamas that overlap)

Exercise 3.1 CountBabaMama

- Skeleton code:

```java
/**
 * Count the number of occurrences of substrings "baba" or "mama"
 * in the input string recursively. They may overlap.
 * For example, countBabaMama("aba babaa amama ma") → 2,
 * and countBabaMama("babababamamama") → 4.
 * @param input is the input string.
 * @return the number of occurrences.
 */
public static int countBabaMama(String input) {


}
```

Exercise 3.2 DelDuplicate

- Given an input string, delete *recursively* all the ***duplicate adjacent characters*** and return a string where all those adjacent characters that are the same have been reduced to just a ***single*** character.
  Do **not** use any loops.
  Do **not** use any regular expressions and methods such as matches, split, replaceAll.

- Test case 1:
  delDuplicate("aaabbc") → "abc"

- Test case 2:
  delDuplicate("aaaaa") → "a"

Exercise 3.2 DelDuplicate

- Skeleton code:

```
/**
 * Remove adjacent duplicate characters in a string.
 * For example, delDuplicate("aaabbc") → "abc".
 * @param input is the input string.
 * @return the resulting string.
 */
public static String delDuplicate(String input) {


}
```

Exercise 3.3 ExtractVowel

- Given an input string, ***complete a helper method*** that extracts the vowels and returns the string of the vowels in the input string ***recursively***.
  All strings are **lowercase**.
  Do **not** use any loops or regular expressions.

- Test case 1:
  extractVowel("i love you 3000") → "ioeou"

- Test case 2:
  extractVowel("aiueo") → "aiueo"

Exercise 3.3 ExtractVowel Public and isVowel Methods

- Assume you are given two methods, one that calls a recursive helper method, and one that gives you true iff the input char is a vowel, that you can use.

```
/**
 * Extract the vowels from the input string.
 * For example, extractVowel("i love you 3000") → "ioeou".
 * @param str is the input string.
 * @return the vowels of the input string.
 */
public static String extractVowel(String str) {
    return extractVowelHelper(str, 0, "");
}

private static boolean isVowel(char c) {
    return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
}
```

Exercise 3.3 ExtractVowel Private Recursive Helper Method

- Your task is to complete the following recursive helper method, where
  - str is the original string
  - start is the starting index to look for the vowels
  - vowels store the previously found vowels

```
private static String extractVowelHelper(String str, int start, String vowels) {
    // base case



    // recursive step



}
```

Exercise 3.4 SkipSum

- Given a list of integers, we want to know whether it is possible to choose a subset of some of the integers, such that the integers in the subset adds up to the given sum ***recursively***.
  We also want that if an integer is chosen to be in the sum, the integer **next to it** in the list must be skipped and **not** chosen to be in the sum.
  Do **not** use any loops or regular expressions.

- Test cases:
  - skipSum([2, 5, 10, 6], 12) → true (since we can choose {2, 10})
  - skipSum([2, 5, 10, 6], 7) → false (since we cannot choose {2, 5})
  - skipSum([2, 5, 10, 6], 16) → false (since we cannot choose {10, 6})

Exercise 3.4 SkipSum

- Given code:

```java
/**
 * Decide whether there is a subset in the input list
 * that adds up to the target sum, where adjacent integers
 * in the list must not be both in the subset.
 * For example, skipSum([2, 5, 10], 12) → true,
 * and skipSum([2, 5, 10], 7) → false.
 * @param list is the input list.
 * @param sum is the target sum.
 * @return true iff there is a subset of non-adjacent integers
 * in the list that adds to sum.
 */
public static boolean skipSum(List<Integer> list, int sum) {

    // call your recursive helper method
    return skipSumHelper(list, 0, sum);
}
```

Exercise 3.4 SkipSum Helper Method

- Skeleton code:

```java
private static boolean skipSumHelper(List<Integer> list, int start,
int sum) {

    // base case



    // recursive call



}
```

Exercise 3.5 OddAndTen

- Given a list of integers, you want to know whether it is possible to divide the integers into two sets, so that the sum of one set is *odd*, and the sum of the other set is a *multiple of 10*.
  Every integer **must be in** one set or the other.
  You ***can*** write a *recursive helper method* that takes ***any number of arguments*** and then call it inside the method, but you ***cannot*** use any loops.

- Test cases:
  - oddAndTen([5, 5, 3]) → true (the sets are {3} and {5, 5})
  - oddAndTen([5, 5, 4]) → false (since we cannot choose odd and mult of 10)
  - oddAndTen([5, 5, 4, 1]) → true (the sets are {5} and {5, 4, 1})

Exercise 3.5 OddAndTen

- Skeleton code 1:

```
/**
 * Decide if it is possible to divide the integers in a list into two sets,
 * so that the sum of one set is odd, and the sum of the other set is a mult of 10.
 * Every integer must be in one set or the other.
 * For example, oddAndTen([5, 5, 3])  →  true,
 * and oddAndTen([5, 5, 4])  →  false.
 * @param list is a list of integers.
 * @return true iff there is one odd partition and the other multiple of 10.
 */
public static boolean oddAndTen(List<Integer> list) {

    // call your recursive helper method



}
```

Exercise 3.5 OddAndTen Helper Method

- Skeleton code 2:

```java
private static boolean oddAndTenHelper(...) { // add any parameters

    // base case



    // recursive call



}
```

Thank you for your attention !

- In this lab, you have learned:
  - Creating good test cases
    - Divide into subdomains, one test case for each subdomain
    - Include the boundaries
  - Solving problems using Recursion + Helper Method