# Recitation 8

# Overview

- Encapsulation
- Recursion
  - Basics
  - Numbers
  - Strings

Recitation 8

# Encapsulation

# Encapsulation (IE Privacy Settings)

- One of the key aspects of programming is to write **secure code**, and allow access only to needed values and functions.
- We can prevent access to some code we can replace the word 'public' with the word 'private'.
- This will be

# Encapsulation - Question

```
public class PizzaPlace {
     private int cashier = 100;
          public void sellPizza() {
               System.out.println(“Thanks for buying a pizza”);
               updateBalance(50);
          }
          private void updateBalance(int amount) {
               cashier += amount;
          }
}
```

```
public static void main (String[] args) { // NOT in the PizzaPlace class
          PizzaPlace.sellPizza();  // ?
          PizzaPlace.updateBalance(-100000); // ?
          PizzaPlace.cashier = 0; // ?
}
```

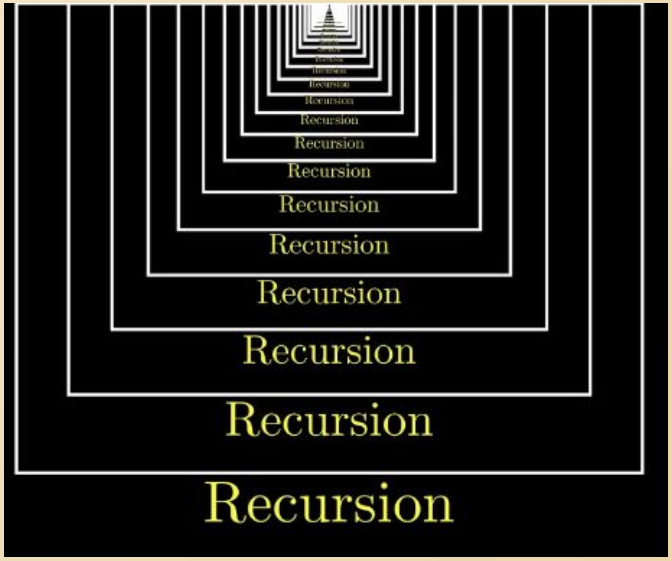
# Encapsulation - Question

```
public class PizzaPlace {
     private int cashier = 100;
          public void sellPizza() {
               System.out.println("Thanks for buying a pizza");
               updateBalance(50);
          }
          private void updateBalance(int amount) {
               cashier += amount;
          }
}
```

```
public static void main (String[] args) { // NOT in the PizzaPlace class
          PizzaPlace.sellPizza();  // Works as expected.
          PizzaPlace.updateBalance(-100000); // ERROR: Can't access private method.
          PizzaPlace.cashier = 0; //  ERROR: Can't access private variable.
}
```

Recitation 8

# Recursion



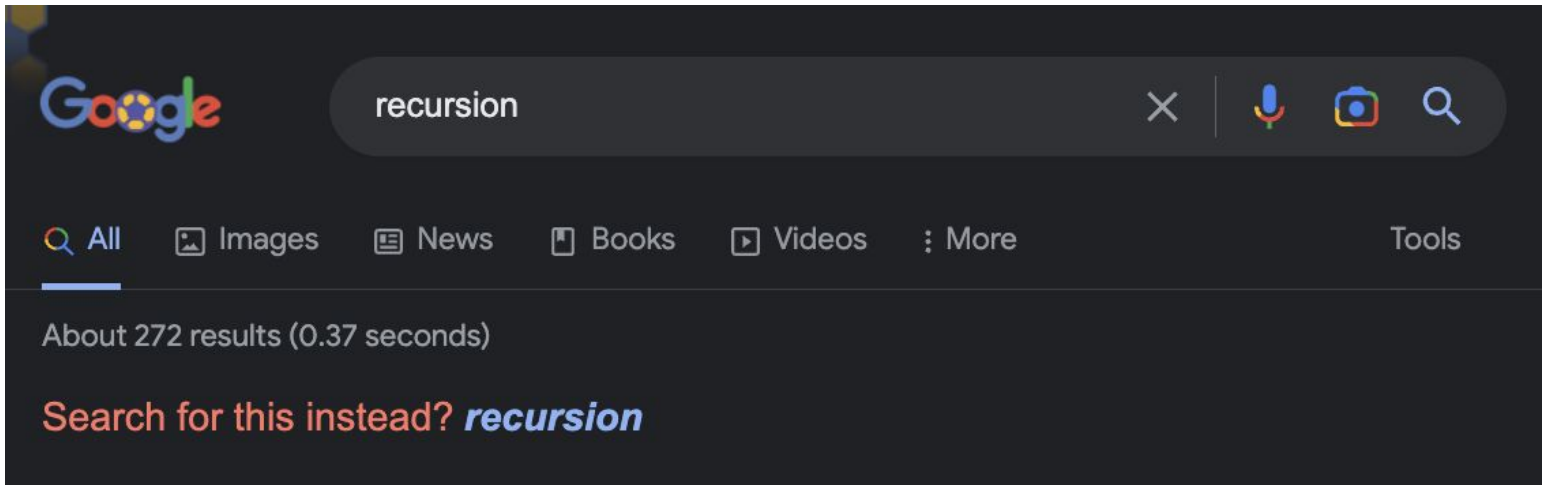
# Recursion

- Recursive function is a function defined with the terms of itself.
- Questions:
  - factorial
    - n! = n * (n-1)!
    - f(n) = n * f(n-1)
  - fibonacci
    - next = previous + current
    - f(n+1) = f(n-1) + f(n)

In programming -> A recursive function is a function which calls itself.

# Recursion - Important Terminology

- Base Case\Stopping condition - The subproblem which I can solve without using any complicated operations.
- Recursive Call - This is the part of the recursive function in which the function calls itself with slight reduction.



# Recursion

- Necessary conditions for a recursive function:
  - There is **at least** one base case which the recursion stops once it reached.
  - Recursive calls to the function are made to solve a "smaller" problem, which makes us closer to the base case of the original problem.
  - The function needs to use the smaller solution to solve the original problem.
  - Faith – lots of it.

# Question 1 - Recursion

- Many recursive functions can be also written as iterative functions
- Let's look at a function we know, factorial.
- Try and find the similarities. There are 6 total..

Iterative Version

```
public static long factorial(int n) {
     if (n <= -1) {
       return -1;
     }
     if (n <= 1) {
       return 1;
     }
     long res = 1;
     for (int i = 1; i <= n; i++) {
       res *= i;
     }
     return res;
}
```

Recurisve Version

```
public static long factorial(int n) {
     if (n <= -1) {
       return -1;
     }
     if (n <= 1) {
       return 1;
     }
     long res = n * factorial(n - 1);
     return res;
}
```

# Question 1 - Recursion

Iterative Version

```
public static long factorial(int n) {
     if (n <= -1) {
         return -1;
     }
     if (n <= 1) {
        return 1;
     }
     long res = 1;
     for (int i = 1; i <= n; i++) {
        res *= i;
     }
     return res;
}
```

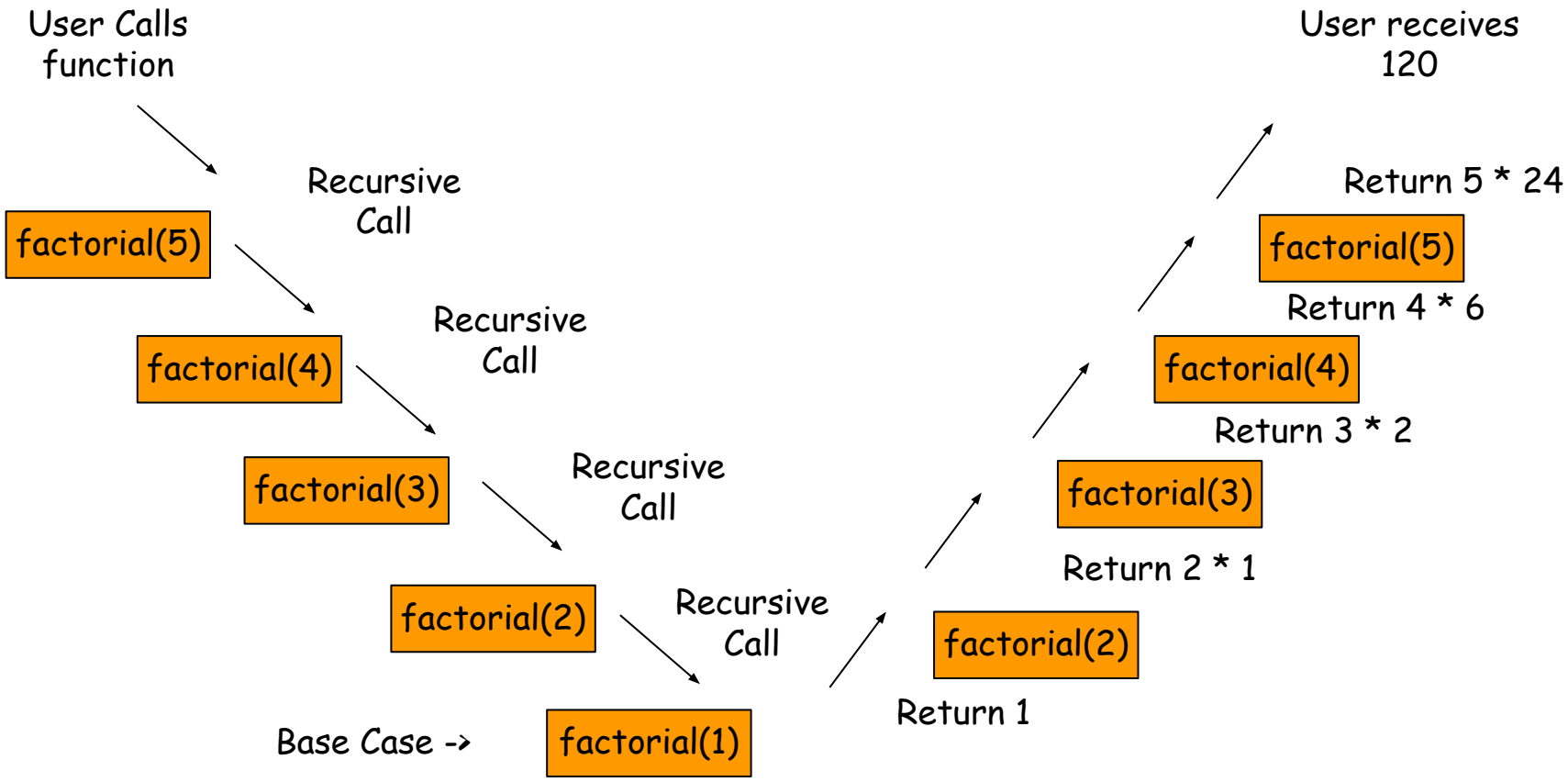
Recurisve Version

```
public static long factorial(int n) {
     if (n <= -1) {
         return -1;
     }
     if (n <= 1) {
        return 1;
     }
     long res = n * factorial(n - 1);
     return res;
 }
```

1. Functions signature & params
2. Error control
3. Base case
4. Calculation
5. Initialize return variable
6. return statement

# Question 1 - Visual Recursion

How does recursion work?



# Question 2 - Recursion

- Not all functions are easy to build as iterative functions way, and the code may be cumbersome.
- One of the main problems recursive function may arise is endless recursion therefore, we must define our stopping condition well.
- Let see the following question and understand this a bit better:

- Fibonacci series is a series of numbers which every element equals to the sum of the two previous elements.
- fib(0) = 0 , fib(1) = 1. fib(2) = 1, fib(3) = 2
- fib(n) = fib(n-1) + fib(n-2)

- Build a recursive function (without loops), which receives an int n, and the return the nth element in Fibonacci series.

# Question 2 - Recursion

```
public static int fibo(int n) {
    if (n == 0) {
        return 0;
    }
    return fibo(n-1) + fibo(n-2);
}
```

- Ok, lets see if it works..
- fibo(0) -> 0 => OK
- fibo(1) -> fibo(0) + fibo(-1) -> 0 + (fibo(-2) + fibo(-3)) ->
  ((fibo(-3) + fibo(-4)) + (fibo(-4) + fibo(-5))) -> …

- What happened logically? I am trying to rely on nonexistent base case.
- What happened code wise? The run is entering an endless recursive function calls, which is very similar to endless loop, however, unlike them it will lead to StackOverFlowException (Run Time Error),
- Let's fix it, by well defining the base case.

# Question 2, Expansion 1 - Recursion

```
public static int fibo2(int n) {
    if (n == 0 || n == 1) {
        return n;
    }
    return fibo2(n - 1) + fibo2(n - 2);
}
```

- fibo(0) -> 0 => OK
- fibo(1) -> 1 => OK
- fibo(2) -> fibo(0) + fibo(1) -> 0 + 1 => OK
- fibo(3) -> fibo(1) + fibo(2) -> 1 + fibo(0) + fibo(1) -> 1 + 0 + 1 -> 2 => OK
- I never enter an undefined value.

# Question 3 – Math Pow

- Let do another function
- We know that
- n^m = n * n * n * n * n * … * n. (m times)
- n^0 = 1

- Build a recursive function (without loops), which receives a double base, and int exponent, and returns base^exponent.

# Question 3 – Math Pow

```
public static double pow(double base, double exp) {
    if (base == 0 || base == 1) {
        return base;
    }
    if (exp == 0) {
        return 1;
    }
    return base * pow(base, exp - 1);
}
```

- Ok, lets see if it works..
- pow(5, 0) -> 1 => OK
- pow(5, 1) -> 5 * pow(5 ,0) -> 5 => OK
- pow(5, -1) -> 5 * pow(5 ,-2) -> 5 * 5 -> pow(5, -3) -> ….

We will use the equivelany -> base^exp = (1/base)^(-exp)

# Question 3, Expansion 1 – Math Pow

```
public static double pow(double base, int exp) {
    if (base == 0 || base == 1) {
        return base;
    }
    if (exp < 0) {
        return pow(1 / base, exp * (-1));
    }
    if (exp == 0) {
        return 1;
    }
    return base * pow(base, exp - 1);
}
```

- Ok, lets see if it works..
- pow(5, 0) -> 1 => OK
- pow(5, 1) -> 5 * pow(5 ,0) -> 5 => OK
- pow(5, -1) -> pow(0.5 ,1) -> 0.2 => OK

# Question 3 - Recursion

```
public static int fibo(int n) {
    if (n == 0) {
        return 0;
    }
    return fibo(n-1) + fibo(n-2);
}
```

- Ok, lets see if it works..
- fibo(0) -> 0 => OK
- fibo(1) -> fibo(0) + fibo(-1) -> 0 + (fibo(-2) + fibo(-3)) ->
  ((fibo(-3) + fibo(-4)) + (fibo(-4) + fibo(-5))) -> …

- What happened logically? I am trying to rely on nonexistent base case.
- What happened code wise? The run is entering an endless recursive function calls, which is very similar to endless loop, however, unlike them it will lead to StackOverFlowException (Run Time Error),
- Let's fix it, by well defining the base case.

# Question 4 – strEquals

- Up to this point we reduced our problem with a counter which is reduced by a certain amount until we reach the base case.
- But what happens when I want to involve Strings?
- **Our base case** is usually the case when we have **an empty string**
- Otherwise, we can say the string is not empty, hence we have at least 1 char, in position 0, and we reduce our problem by using substring(1), to do the recursive call.
- Let look build a recursive version for str.equals(str1).

# Question 4 - Solution

```
public static boolean stringEquals(String str1, String str2) {
     if (str1.length() != str2.length()) {
           return false;
     }
     // we may assume the strings length are equal (from here)
     if (str1.length() == 0) { // we finished going over the strings
           return true;
     }
          if (str1.charAt(0) != str2.charAt(0)) {
           return false;
     }
     return stringEquals(str1.substring(1), str2.substring(1));
}
```

# Question 5 – charRunCount

- The function charRunCount receives a String and a char and return the number of appearances of the given char in the String:
  - charRunCount("hello",'l'); // 2
  - charRunCount("helLo",'l'); // 1
  - charRunCount("hello world",'l'); // 3
  - charRunCount("hello world",'m'); // 0
- Build a recursive version of that function.

# Question 5 – Solution

```
public static int charRunCount (String str, char ch){
        if (str.length() == 0){
            return 0;
        }
        if (str.charAt(0) == ch){
            return 1 + charRunCount(str.substring(1), ch);
        }
        return charRunCount(str.substring(1), ch);
}
```

# Question 6 - Hamming Distance

- Hamming distance is defined as in how many chars differentiate between 2 words.
- The function hammingDistance receives a couple of Strings consist of lowercase only letters and returns their hamming distance. If their length isn't equal return -1.
  - hammingDistance("hello","yellow"); // -1
  - hammingDistance("monkey","donkey"); // 1
  - hammingDistance("key", "way"); // 2
  - hammingDistance("key", "sky"); // 2
- Build a recursive version of that function.

# Question 6 - Solution

```
public static int hammingDistance (String str1, String str2){
        if (str1.length() != str2.length()){
            return -1;
        }
        if (str1.length() == 0){
            return 0;
        }
        if (str1.charAt(0) != str2.charAt(0)){
            return 1 + hammingDistance(str1.substring(1), str2.substring(1));
        }
        return hammingDistance(str1.substring(1), str2.substring(1));
    }
```

# Question 7 - isPalindrome

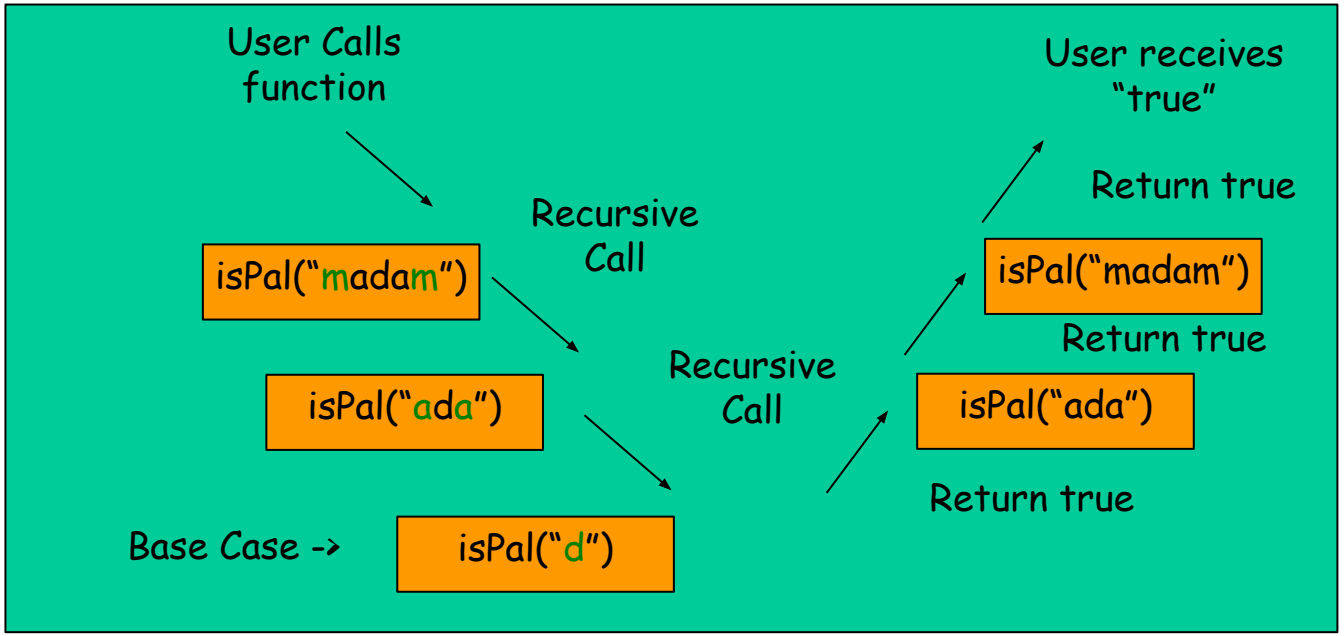
- Sometimes there is more than one base case, as we saw with ints, now lets see one with strings.
- Furthermore, not always removing the first char by using substring(1) is the correct approach.
- Let examine a function who has both in its recursive version: isPalindrome.
- A word is considered a palindrome iff I can read it from left to right AND from right to left and get the same word
  - isPalindrome("bob") -> true
  - isPalindrome("maam") -> true
  - isPalindrome("madam") -> true
  - isPalindrome("barb") -> false

# Question 7 - Solution

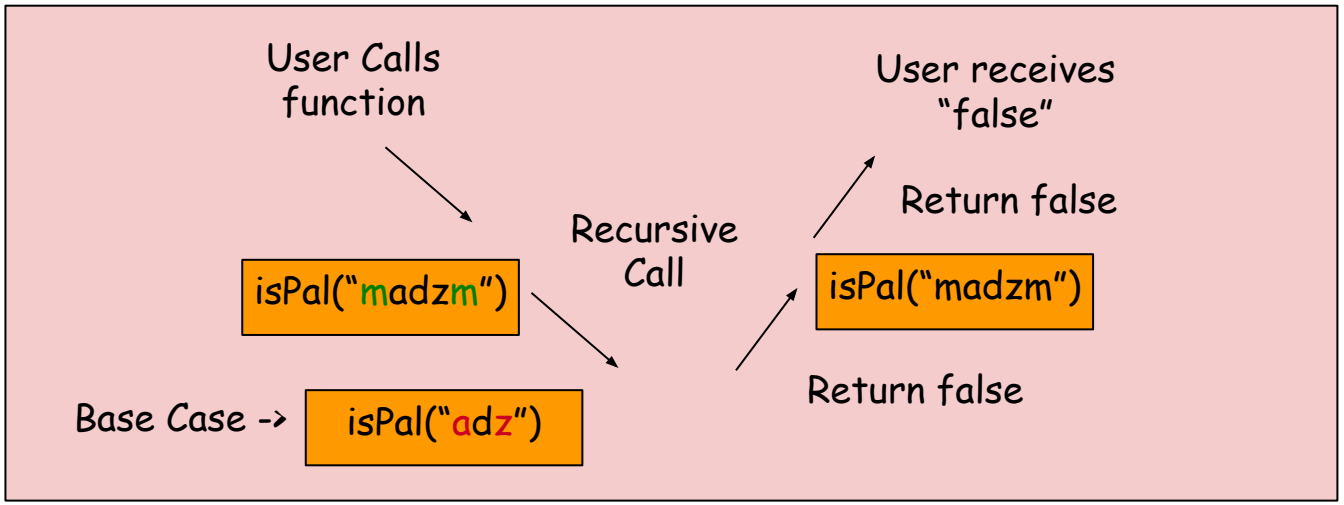
```
public static boolean isPalindrome(String str) {
     if (str.length() <= 1) {
           return true;
     }
     if (str.charAt(0) != str.charAt(str.length() - 1)) {
           return false;
     }
          return isPalindrome(str.substring(1, str.length() - 1));
}
```

# Question 7 - Visual Recursion

"true" input




"false" input



# Question 8 - indexOf

- Sometimes we will want to use helper function to help navigate the function. The helper function allows us to add parameters if needed. When you are using a helper function it also allows you to have another params if needed.
- Its common practice that the helper functions are not public, rather they are private.
- Let's see this in the recursive implementation for str.indexOf(char ch).

# Question 8 – Solution

```
public static int indexOf (String str, char ch) {
     int res = indexOfHelper(str,ch);
     if (res == str.length()) {
          return -1;
     }
          return res;
}
private static int indexOfHelper (String str, char ch) {
     if (str.length() == 0 || str.charAt(0) == ch) {
          return 0;
     }
          return 1 + indexOfHelper(str.substring(1),ch);
}
```

# Question 9 - ASCII

- Now, lets combine ASCII with recursion.
- Let see a function you are already saw stringToInt and build it recursive version.

# Question 9 - Solution

```
public static int stringToInt(String num) {
     if (num.length() == 0) {
           return 0;
     }
     int number = (int) (Math.pow(10, num.length() - 1) * (num.charAt(0) - '0'));
     return stringToInt(num.substring(1)) + number;
}
```

# Question 10 – Anagram, Finals 2019

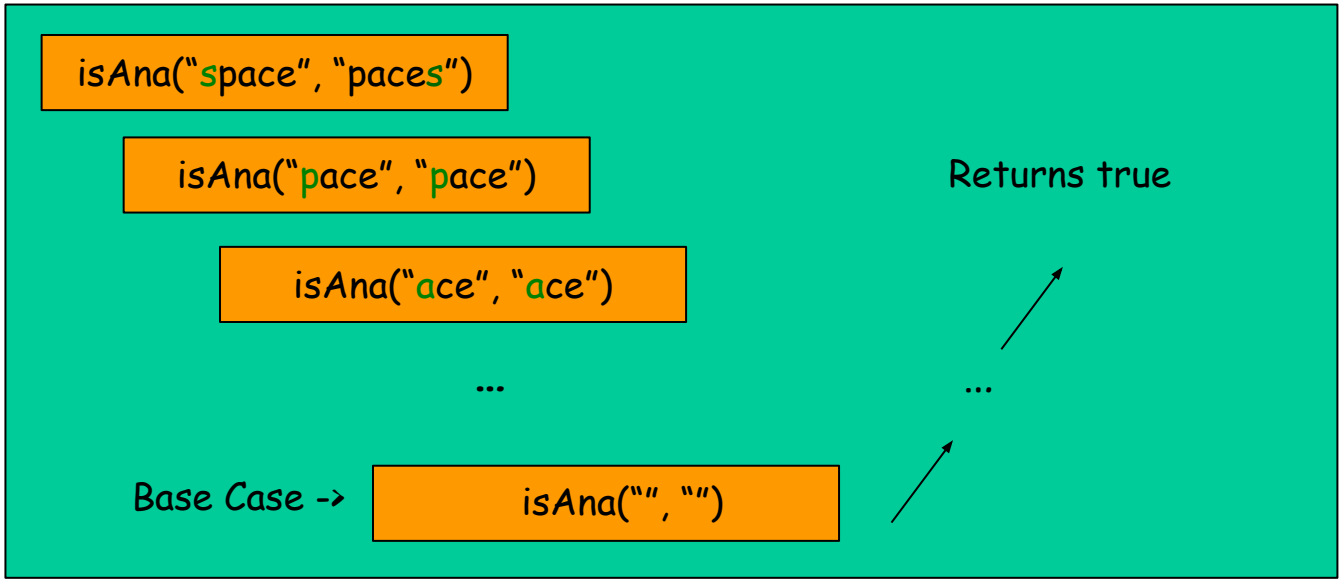
- Let look at a question which also reduces the string in a different way: anagram, in this version of the question we will assume there are no non alphabet chars within the string.
- reminder:
  - isAnagram("car","car") -> true
  - isAnagram("car","arc") -> true
  - isAnagram("monkey","donkey") -> false
  - isAnagram("space","paces") -> true
  - isAnagram("spaces","paces") -> false

# Question 10 - Solution

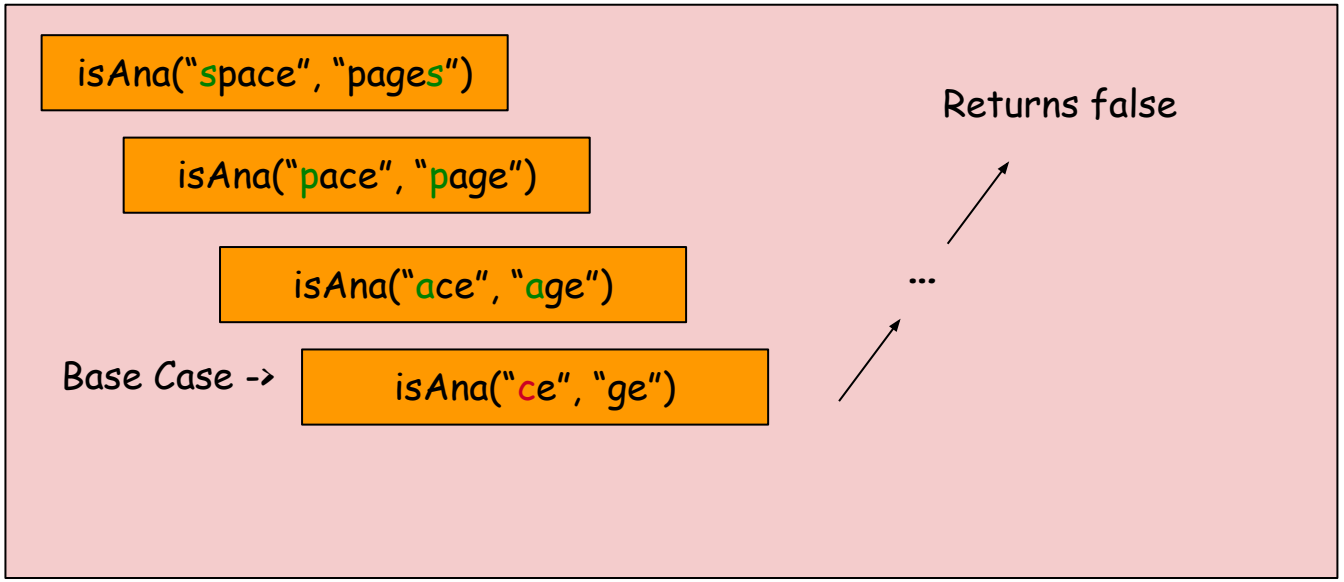
```
public static boolean isAnagram (String str1, String str2) {
     if (str1.length() != str2.length()){
          return false;
     }
     if (str1.length() == 0){
          return true;
     }
     int index = str2.indexOf(str1.charAt(0));
     if (index == -1){
          return false;
     }
     String temp = str2.substring(0, index);
     if (index != str2.length() - 1) {
          temp += str2.substring(index + 1);
     }
     return isAnagram(str1.substring(1), temp);
}
```

# Question 10 - Solution

"true" input




"false" input



# Question 11 - Valid Parentheses – Finals 2022A, Q2

- Given a String str which consists of only the following characters: '(' ,')' ,'0'…'9' ,'-' ,'+' ,'*' ,'/' ,'%'. In other words, str contains digits, basic math operations, and rounded parentheses characters only. Write a **recursive** function which receives str and returns true if the str has a legal assignment of parentheses, false otherwise. A legal assignment of parentheses means that any opening parenthesis, has a closing parenthesis counterpart somewhere down the string and no closing parentheses is opened before there is a open parentheses.

  String input1 = "()";

  String input2 = "(";

  String input3 = "((()))";

  String input4 = "(()";

  String input5 = ")(";

  - System.out.println(isValidParentheses(input1));// true
  - System.out.println(isValidParentheses(input2));// false
  - System.out.println(isValidParentheses(input3));// true
  - System.out.println(isValidParentheses(input4));// false
  - System.out.println(isValidParentheses(input5));// false

# Question 11 – Solution

```
public static boolean isValidParentheses (String s) {
        return isValidParentheses(s, 0);
}

private static boolean isValidParentheses (String str, int counter){
        if (counter < 0) {
            return false;
        } // if the counter is negative, return false (reached end without open)
        if (str.length() == 0) {
            return counter == 0; // if the string is empty, return true if the counter is 0, otherwise, false
         } // if the first character is not a parenthesis, call the function again with the rest of the string, and
       the same counter
        if ("()".indexOf(str.charAt(0)) == -1) {
            return isValidParentheses(str.substring(1), counter);
        }
       /* if the first character is a parenthesis, call the function again with the rest of the string
        the counter will increase or decrease by 1, depending on the parenthesis
        if the parenthesis is '(' the counter will increase by 1,
        if the parenthesis is ')' the counter will decrease by 1.
        "))("indexOf(')') will result in 0 ==> 0 - 1 = -1
        "))("indexOf('(') will result in 2 ==> 2 - 1 = 1
             */
        return isValidParentheses(str.substring(1), counter + ("))(".indexOf(str.charAt(0)) - 1));
}
```