## Activity 1 (Portfolio Task):

Implement classes **Student**, **Module** and **StudentSupportOffice** that provide the following functionality:

* Class **Student**:
  + Stores **student id**
  + Creates a **human friendly representation** of the class (i.e. **toString()**)
  + Allows a **total ordering** of students (i.e. students are **Comparable**)
* Class **Module**:
  + Stores **module id**, a **class list** (**an array** of enrolled students with **default capacity for 5 students**), and the **current size** of the class list
  + Allows students to be **enrolled** if a student is **not already in the class list** (while maintaining the class list in **ascending order** based on **student id**)
  + Allows students to be **unenrolled** (the class list **must not have gaps** that “break” the sequence of enrolled students)
  + Creates a **human friendly representation** of the class (i.e. **toString()**, improve print outs by **ignoring empty elements** within the class list)
* Class **StudentSupportOffice**:
  + Creates **three modules**: "CIS2206", "CIS2360" and "CIS2205"
  + Creates **five students**: "U0000001", "U0000002", "U0000003", "U0000004", "U0000005"
  + **Enrols/unenrolls students** to modules as described below

Initially, students enrol to modules:

* Module "CIS2206" contains in its class list:
  + Student "U0000001"
  + Student "U0000005"
* Module "CIS2360" contains in its class list:
  + Student "U0000001"
  + Student "U0000003"
  + Student "U0000004"
* Module "CIS2205" contains in its class list:
  + Student "U0000002"
  + Student "U0000004"
  + Student "U0000005"

Subsequently, students change modules (enrol and unenroll). Finally, modules contain the following students:

* Module "CIS2206" contains in its class list:
  + Student "U0000001"
  + Student "U0000003"
  + Student "U0000004"
* Module "CIS2360" contains in its class list:
  + Student "U0000001"
  + Student "U0000002"
  + Student "U0000003"
  + Student "U0000004"
  + Student "U0000005"
* Module "CIS2205" contains in its class list:
  + Student "U0000001"
  + Student "U0000003"
  + Student "U0000004"
* Extend class Student to allow additional fields such as name, surname and age, while allowing multiple types of comparisons (e.g. sort by either id or surname)
* Allocate dynamically additional memory for Module’s class list when current capacity is exhausted (each time double the size of current class list array) and copy existing students into the new class list
* Add JUnit testing to ensure that no errors were introduced during development
* Consider the Complexity Analysis of implemented methods

**What to include in your Portfolio:**

* **Report:** Describe in your report which parts of **Activity 1** have been successfully implemented
* **IntelliJ Project:** Include your Java code for **Activity 1** in your IntelliJ project under package "Practical\_11"

## Activity 1 (Portfolio Task 2):

Implement classes **Team**, **FootballLeague**, and **Dashboard** that provide the following functionality:

* Class **Team**:
  + Stores team’s **name**, number of **wins**, **draws** and **losses**, as well as the number of **points** (**points** are **recalculated** every time the number of wins, draws and losses is updated). **Note:** A team is awarded **3 points** for every **win**, **1 point** for every **draw** and **0 points** for every **loss**
  + Creates a **human friendly representation** of the class (i.e. **toString()**)
  + Allows a **total ordering** of teams (i.e. teams are **Comparable**)
* Class **FootballLeague**:
  + Stores a **list** of Teams included in the league (while maintaining the list in **descending order** based on **Teams’ points**)
  + **Teams** can be **inserted** (adding a new team) and **removed** (deleting an existing team) at any point
  + **Team’s position** in the league can be **updated** by updating team’s wins, draws and losses (**changes** to points **must be reflected** in the **ordering** **of the list**)
  + Teams can be **relegated** at the end of the season (i.e. the **3 teams** with the **least amount of points**)
  + The **league table** (i.e. list of Teams) should be **printed** **after every change** takes place
* Class **Dashboard**:
  + Creates a **football league instance** and subsequently **inserts**, **updates**, **removes** and **relegates** teams

## Activity 2 (Optional Task):

Improve your implementation of ‘Practical 12 - Activity 1’ by extending it:

* Add JUnit testing to ensure that no errors were introduced during development
* Consider the Complexity Analysis of implemented methods

## Activity 3 (Optional Task):

Consider further reading for Unit 12:

* Implement a Singly Linked List, where the element within each node is a specific class such as String (instead of using generics)
* Implement a Doubly Linked List, where the element within each node is a specific class such as String (instead of using generics)

**What to include in your Portfolio:**

**Report:** Describe in your report which parts of **Activity 1** have been successfully implemented

* **IntelliJ Project:** Include your Java code for **Activity 1** in your IntelliJ project under package "Practical\_12"

## Activity 1 (Portfolio Task 3):

We are used to mathematical expressions such as “( (4 + 5) - (1 \* 2) ) / 5”, also known as infix notation, where operators are used in-between operands. However, for a calculator it is easier to process a postfix notation such as “4 5 + 1 2 \* - 5 /”, where the operator is written after the operands.

We can use a stack to evaluate the postfix expression “4 5 + 1 2 \* - 5 /” (its infix counterpart is “( (4 + 5) - (1 \* 2) ) / 5”). The state of the stack after each token (i.e., an operator or operand) is processed is depicted below:

| Method | **Return Value** | **Stack Contents ( … → top)** |
| --- | --- | --- |
| **push(4)** | 4 | (4) |
| **push(5)** | 5 | (4,5) |
| **push(+)** | + | (4,5,+) |
| **pop()** | + | (4,5) |
| **pop()** | 5 | (4) |
| **pop()** | 4 | () |
| **push(9)** | 9 | (9) |
| **push(1)** | 1 | (9,1) |
| **push(2)** | 2 | (9,1,2) |
| **push(\*)** | \* | (9,1,2,\*) |
| **pop()** | \* | (9,1,2) |
| **pop()** | 2 | (9,1) |
| **pop()** | 1 | (9) |
| **push(2)** | 2 | (9,2) |
| **push(-)** | - | (9,2,-) |
| **pop()** | - | (9,2) |
| **pop()** | 2 | (9) |
| **pop()** | 9 | () |
| **push(7)** | 7 | (7) |
| **push(5)** | 5 | (7,5) |
| **push(/)** | / | (7,5,/) |
| **pop()** | / | (7,5) |
| **pop()** | 5 | (7) |
| **pop()** | 7 | () |
| **push(1.4)** | 1.4 | (1.4) |

Implement class **Calculator**, which:

* Takes as input a postfix expression (e.g. "4 5 + 1 2 \* - 5 /")
* Processes operators and operands (of the given postfix expression) left to right
* Uses a stack in order to calculate the postfix expression (as depicted above)
* Prints output that recreates the abovementioned table (i.e. columns “Method”, “Return Value” and “Stack Contents”

## Activity 2 (Optional Task):

Improve your implementation of ‘Practical 13 - Activity 1’ by extending it:

* Add JUnit testing to ensure that no errors were introduced during development
* Consider the Complexity Analysis of implemented methods

## Activity 3 (Optional Task):

Consider further reading for Unit 13:

* Implement an array-based stack, where the element of the array is a specific class such as String (instead of using generics)
* Implement a stack using your implementation of lists from ‘Practical 12 -   
  Activity 3’

**What to include in your Portfolio:**

* **Report:** Describe in your report which parts of **Activity 1** have been successfully implemented
* **IntelliJ Project:** Include your Java code for **Activity 1** in your IntelliJ project under package "Practical\_13"

## Activity 1 (Portfolio Task 4):

Implement classes **PhoneBookEntry**, **PhoneBook** and **User** that provide the following functionality:

* Class **PhoneBookEntry**:
  + Stores details of a phone book entry, including **full name** (i.e. both surname and name), **phone number**, **email**, and **address**
  + Creates a **human friendly representation** of the class (i.e. **toString()**)
* Class **PhoneBook**:
  + Stores **phone book entries in a map** based on their details, where details are mapped as: **“full name” → “entry”**, namely phone book entries are stored in map of type:   
    **Map<String, PhoneBookEntry>** (note that **duplicates** of full names are **not allowed**)
  + **Phone book** entries can be **inserted** (an entry for a **new full name**), **updated** (updating an entry for an **existing full name**) or **deleted** (deleting an entry for an **existing full name**)
  + **Prints** the details of an **existing entry** (**if present** in the phone book) by **searching** the phone book **based on a given full name** (thus showing the corresponding number)
  + **Prints all entries** within the phone book
* Class **User**:
* Creates a **phone book instance** and subsequently **inserts**, **updates**, **removes**, and **prints** phone book **entries**

## Activity 2 (Optional Task):

Improve your implementation of ‘Practical 17 - Activity 1’ by extending it:

* Allow duplicate names by uniquely identifying each entry based on address, namely use a map within a map that maps entries as: “full name” → “address” → “entry” (i.e. Map<String, Map<String, PhoneBookEntry>>)
* Allow a further map that maps entries as: “number” → “entry” (namely users can search for entries based on a given number)
* Add JUnit testing to ensure that no errors were introduced during development
* Consider the Complexity Analysis of implemented methods

## Activity 3 (Optional Task):

Consider further reading for Unit 17:

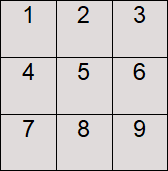
* Implement a Map, where the key and value within each entry are of a specific class such as String and Integer (instead of using generics)
* Implement the WordCount example of ‘Chapter 10.1.2 Application: Counting Word Frequencies’ using your implementation of a Map<String, Integer>

**What to include in your Portfolio:**

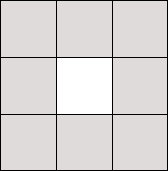
* **Report:** Describe in your report which parts of **Activity 1** have been successfully implemented
* **IntelliJ Project:** Include your Java code for **Activity 1** in your IntelliJ project under package "Practical\_17"

## Activity 1 (Portfolio Task 5):

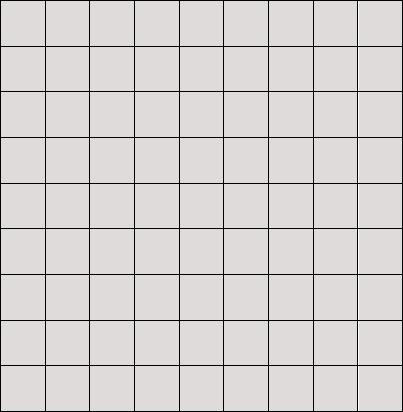
Implement recursively the [Sierpiński carpet](https://en.wikipedia.org/wiki/Sierpiński_carpet). More specifically, a recursive function will take as input a 2D array of characters (i.e. char[][] board) of size 3n X 3n, where 1 ≤n ≤ 5 (i.e. 3x3, 9x9, 27x27, 81x81, 243x243). The 2D array can be divided into 9 sub-arrays of size 3n-1 x 3n-1, e.g. a 3x3 array can be divided into 9 sub-arrays of size 1x1:



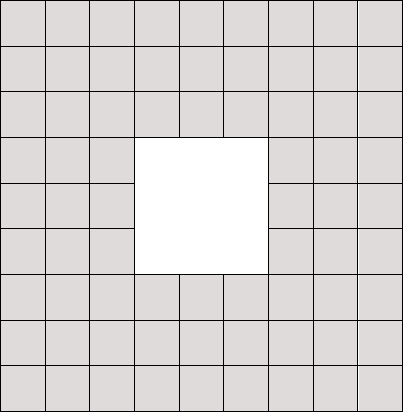
We can now remove the middle sub-array (i.e. sub-array 5) resulting in the following array:



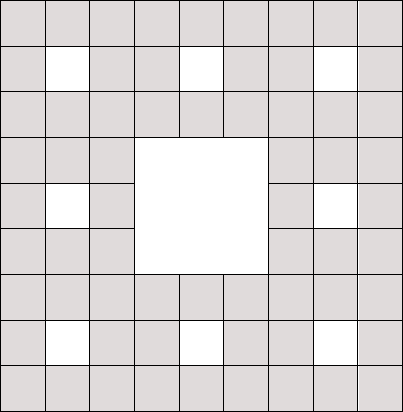
For larger arrays this process is applied recursively, e.g. a 9x9 array:



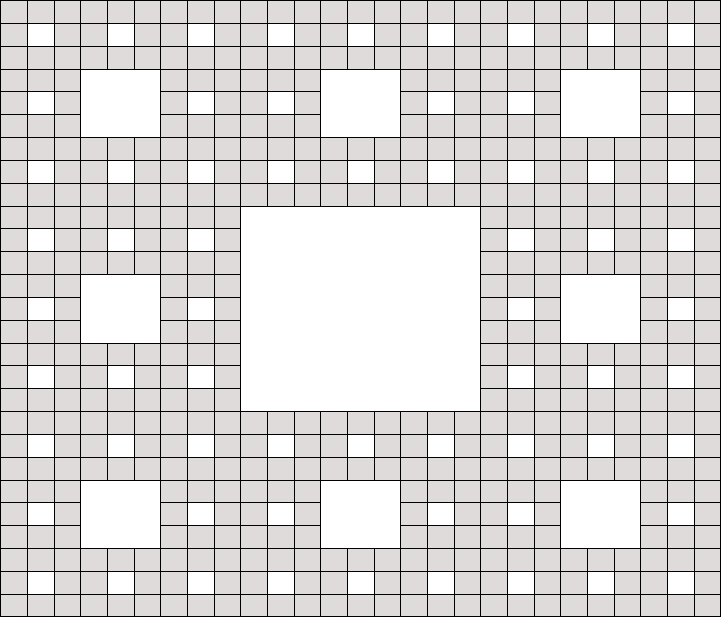
First, remove the middle 3x3 sub-array:



Then, apply the process recursively on the 8 sub-arrays of size 3x3 (removing arrays of size 1x1):



For an array of size 27x27, the Sierpiński carpet would be as follows (remove 9x9 sub-array, then 3x3 sub-arrays, and then 1x1 sub-arrays):



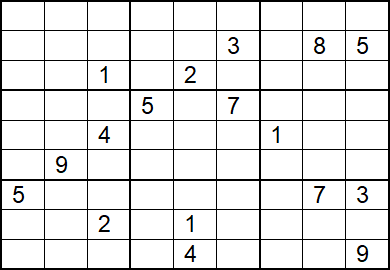
The array or characters is initialised by setting each cell to ‘\*’, while removing a sub-array sets the corresponding cells to ‘ ’. File ‘Sierpinski\_Carpet.zip’ contains a Sierpiński carpet for sizes 3x3, 9x9, 27x27, 81x81 and 243x243 (in case sizes 81x81 and 243x243 are unreadable in your text editor, then you will need to reduce the font).

Implement class **SierpinskiCarpet**, which:

* **Stores a 2D array of characters** (i.e. char[][] board) as well as the **dimension size of the array** (i.e. 3, 9, 27, 81 or 243)
* **Initializes** the **board** **with** characters **‘\*’**
* Calls a **recursive function** that removes sub-arrays (by setting cells to ‘ ’) resulting in a Sierpiński carpet (note that the **recursive function** must take the **2D array as input**, but **can take additional parameters** that facilitate the recursion)
* **Prints** the **Sierpiński carpet**

## Activity 2 (Optional Task):

Implement recursively the [Sudoku puzzle](https://en.wikipedia.org/wiki/Sudoku). The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid contains all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a single solution. The following puzzle is considered difficult to be solved by a brute force algorithm (i.e. an algorithm that tries every possible digit until a solution is found):



**What to include in your Portfolio:**

* **Report:** Describe in your report which parts of **Activity 1** have been successfully implemented
* **IntelliJ Project:** Include your Java code for **Activity 1** in your IntelliJ project under package "Practical\_20"