Game Lab 2 - Modules 3 and 4

This assignment spans the content of modules 3 (Conditionals) and 4 (Loops). Therefore, you can start already in week 2 but you will probably not be able to conclude it until week 3 (aka, you will work on this lab during two tutorial sessions).

Learning goals:

- Writing, compiling and running simple Java program
- Using control flow to create different execution paths
- Using a method to perform repeated tasks

During the course of your game, a user is going to have to make decisions on what they want to do. Based on these, the program should follow a certain path forward.

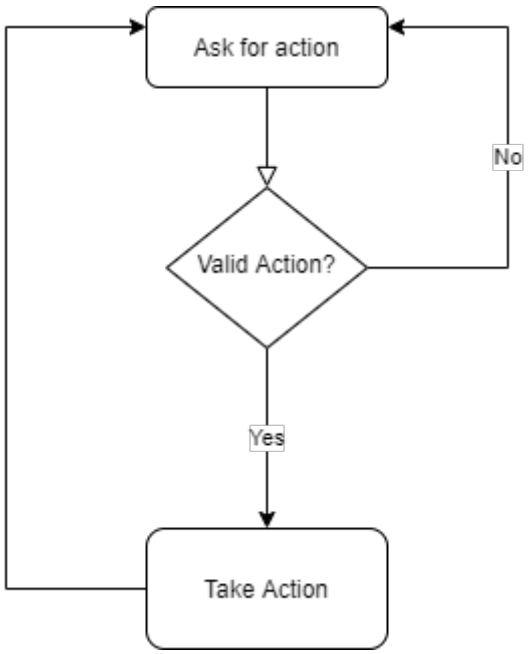
We're going to add a few options to our program. In total, a user can take **8** different actions in the game.

- **Open the door**
- **Go north**
- **Go east**
- **Go south**
- **Go west**
- **Take item**
- **Drop item**
- **Use item**

When a user wants to do something that is not supported, we need to tell them and ask again for a valid input. If the input is valid, the game progresses further and we'll ask the user for new input. At different stages of the game, different inputs can be accepted.

1. Start with last week's `Game.java`. Let's make a new method that asks for a user's input, validates it, and returns it to our main function. After this step your program should have 3 methods:
    a. the `main` method from Game Lab 1.
    b. the `calculateAge` method from Game Lab 1.
    c. the `getInput` method that you will write for this assignment, Game Lab 2.

2. Let's take a look at how the program should run after this assignment:



So, it looks like we'll have to keep asking the user for input. Both if the input was incorrect, but also after taking an action when the input was correct. What does that sound like to you? Right, loops!

3. But before we're there, first write the `getInput` method.
    a. This method uses the `Scanner` class to ask for a user's input.
    b. Next, it checks whether the input is one of the *valid* inputs described above.
    c. It then prints the input and returns it.
    d. If the input is *invalid*, it should ask the user again, and again, and again and again …. until they provide a valid one! Think of what kind of variable you need to control this loop.
    e. What is the condition under which the loop can stop and the function can return?

After you've written your method, call it from the main method, and try it out. Your output should match the output below:

```
...

You are standing in an abandoned university office. There are neither students
nor teachers around you. There's a table in front of you with various papers,
pens, a small puzzle toy, and a calculator.
A large window shows an empty office building; there are no Zombies in the empty
building (as far as you can tell). Behind you is a dark and mysterious door that
leads to a well-lit corridor with a fireproof ceiling and floor. You feel a
sense of Wi-Fi around you, the grinding of an LCD operated coffee machine can be
heard in the distance. You are not thirsty, but you rather have a craving for
justice.
```

```
What would you like to do?: Open the floor
Invalid input
What would you like to do?: Get on the door
Invalid input
What would you like to do?: Everybody walk the dinosaur
Invalid input
What would you like to do?: Open the door
Open the door
```

4. What happens when the user gives "*open the door*" as input instead of "*Open the door*"?
   Is this desirable? If not, maybe you can check the `toLowerCase()` method of the String class. See how you can use this method to make it a bit easier on your players.

   How do you learn about this method and how to use it? Software developers have a special way of figuring out things, they use a site that, given any input, provides the desired output: https://letmegooglethat.com/?q=java+String+toLowercase

   You'll find many resources that will help you along. You have plenty of resources available: user-centered (like StackOverflow), tutorials (like w3schools) or documentation (like the official JDK documentation). Do not be afraid to look up things you do not understand, professional software developers lean heavily on documentation as there is simply too little space in their brains to store all the different methods, classes for all SDKs (Software Development Kits) and APIs (Application Programming Interfaces) in existence!

5. Now the game continues. Based on the user's action we should tell them what happens next. Of course, this depends on the previous *state* the program was in. Meaning, where our player *is* within the game's progression at that moment. At each state of the game, taking different actions lead to different results. We don't want to build a game where nothing happens and the player stays in the same room the entire game doing the same thing, which is boring.

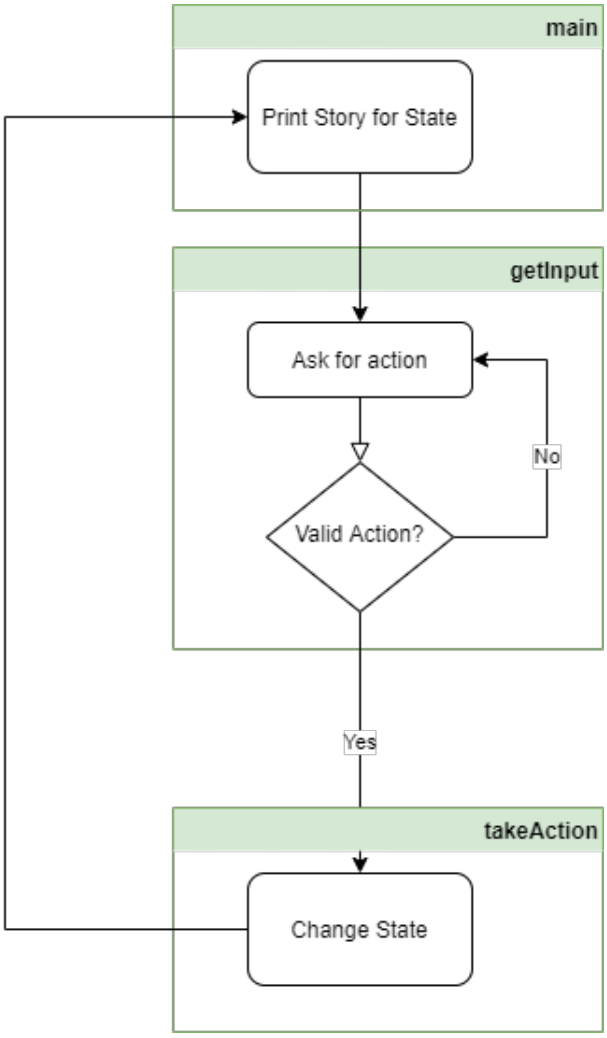
   So let's look at how we can define "*game states*". At the beginning of the game, the player is in Jerry's office. Next, based on the player's action we should go to a new state, ask for an action, go to a new state and so forth until we finish the game. This sounds really complicated, so let's use our programming skills to make it easier.

   First, let's map the states to integers, so the game starts in `stateId 0.` Whenever we take an action, we go to a new state with a new `stateId`.
   State Id, so cool! Now we're really starting to sound like programmers!
   We can now map the `stateId` to our story, i.e. every `stateId` maps to a different part of the story. Make sure that you declare a `stateId` variable in your main method.

6. Now, we need to be able to go from one state to the next, this is called a state transition. Write a method that is responsible for state transitions named `takeAction`. This `takeAction` method has two parameters: `String action` and `int currentState` (in this order). Given these parameters, the method returns a new state.

    a. Write the content of the method, it should check the current state and given an action return a new state.
    Add only the following transitions:
        i. If the current state is 0 and action is “Open the door” the new state is 1
        ii. if the current state is 0 and action is “Take item” the new state is 2
        iii. if the current state is 1 and action is “Go east” the new state is 3
        iv. In all other cases, stay in the current state
    b. Now, alter your main method. Use the `getInput` method to ask the user what to do.
    c. Store the action (which we know is valid), and use it to call the `takeAction` method. Get the new state from the method and store it in the `stateId` variable

7. Now let’s look at the new flow of our program with the corresponding responsible methods:



All that remains, for now, is to:

1. Write the stories for our new `stateId` values, namely 1, 2 and 3
2. Alter the main function so the program keeps asking the user for new actions; the so-called game loop (we will do this in the next step, so keep reading!)

8. Now that we have states, actions and transitions, we need a game loop to make sure the game keeps running while the user is playing. Think about what would happen if we don't use a loop in our game… There would be no game, just a boring form with a single route from start to end, essentially turning your game into meaningless bureaucracy.

   As strange as it may sound, a user might want to quit our fabulous game. So, in this very unlikely case, we have to provide special action.
   Alter your main method so the player can keep writing actions as long as they don't write **`quit`** to exit your game. Move your `getInput` and `takeAction` method calls inside the loop to make sure your game keeps on running.

9. The loop containing the `getInput` and `takeAction` calls must keep running until the user writes **quit**. But how can we tell this to our loop? To do this, let's assign a special `stateId` to exit the game, `stateId` ***666***. Adjust your `getInput` to accept quit as an input and alter `takeAction` to return **666** when the input is quit. Now alter your loop to check the value of `stateId`. The game loop should run as long as the `stateId` is not **666**.

10. Finally, if we want to print the story for each state inside our main method, it's gonna get messy! So write another method `printState` with a parameter `int stateId` that we can use to print the story for any given `stateId`. You can print the following stories for each stateId:

    0:

    You are standing in an abandoned university office. There are neither students nor teachers around you. There's a table in front of you with various papers, pens, a small puzzle toy, and a calculator.
    A large window shows an empty office building; there are no Zombies in the empty building (as far as you can tell). Behind you is a dark and mysterious door that leads to a well-lit corridor with a fireproof ceiling and floor. You feel a sense of Wi-Fi around you, the grinding of an LCD operated coffee machine can be heard in the distance. You are not thirsty, but you rather have a craving for justice.

    1:

    You are in a long hallway. There's a man wearing glasses at the end of it, he looks harmless. West is a wall, east is the man, to the north is nothing but empty offices, a desperate sight. The carpeting in the hallway feels soft, you hear the clicking of a mouse in the distance. Your office is south (behind you).

2:

You take the calculator from your desk. It's a Casio FX-85gt Plus. The display shows the number 0.1134. You turn it upside down; now the Casio greets you with a friendly "hello", nice. You hold the calculator in your hand.

3:

The man greets you and starts endlessly talking to you about his children and his holiday to Benidorm. You die of boredom.

For testing purposes, make sure your output matches exactly the stories above for each state.

11. Make sure your game can be quit. If quitting does not work, you can always stop the execution from Visual Studio Code or press *CTRL+c* on your keyboard.
12. Once you are happy with your result and it behaves like described above, save your code to a file named `Game.java` and store it in a safe location. You will need it in the upcoming tutorials!