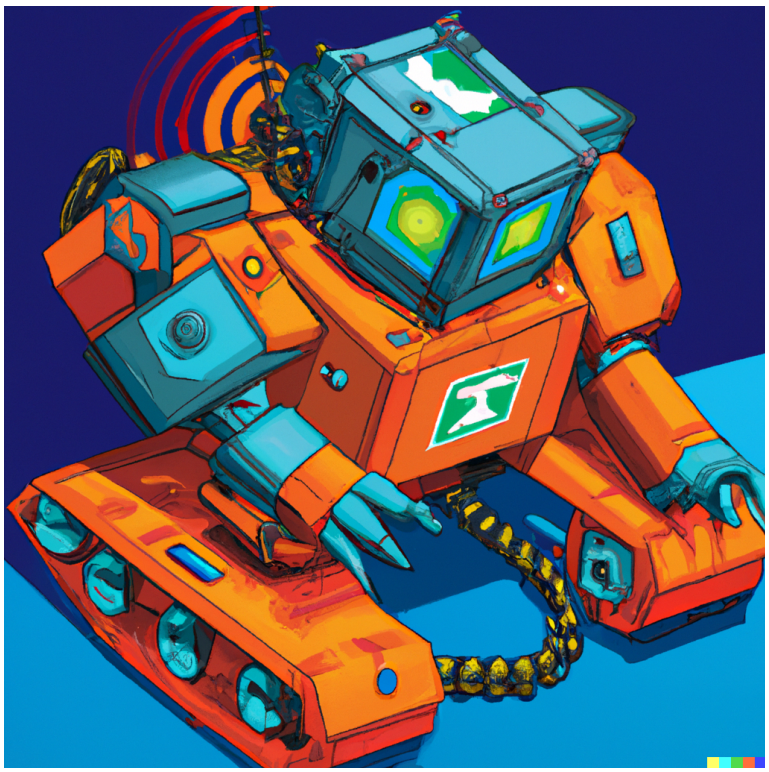
Final Project: Disaster Relief Robots

Introduction

You are a software developer at **RescueBots Inc.**, a robotics company that develops technology to aid in life-threatening situations. The company has recently built a fleet of autonomous robots that can assist in rescue operations during natural disasters, such as bushfires and floods. RescueBots' robots can navigate unknown terrain and successfully locate and rescue people in need of help. For these robots to be truly autonomous and effective, however, they must be capable of making difficult decisions. The RescueBots robots must be programmed to prioritize and determine the order of rescue operations, a feat with ethical and moral implications.

In this project, you will create the decision engine for RescueBots, i.e., a program designed to explore different scenarios and make critical decisions about whom to save. You will finally audit your decision-making algorithm through simulations, and allow users of your program to judge the outcomes themselves.



Academic Honesty

- All assessment items (assignments, test and final project) must be your own, individual, original work.
- Any code that is submitted for assessment will be automatically compared against other students' code and other code sources using sophisticated similarity checking software.
- Cases of potential copying or submitting code that is not your own may lead to a formal academic misconduct hearing and prosecution.
- Potential penalties can include getting zero for the project, failing the subject, and even expulsion from the university in extreme cases.
- For further information, please see the university's Academic Integrity website or ask your teaching team.

Pre-amble: Object Oriented Design

In this final project, you will need to follow best practices of Object-Oriented Programming as taught and discussed throughout the semester.

When creating packages and classes, carefully consider which members (classes, instance variables or methods) should go where, and ensure you are properly using polymorphism, inheritance, and encapsulation in your code.

One class has been provided to you in the starter code (**RescueBot.java**), which *must* be used as your program's entry point. You will need to add additional classes and possibly a package so as to make your code more modular.

This project requires a significant amount of time so make sure you plan accordingly! Time management is of the essence when developing software. Start early, don't leave it to the last week to fire up your IDE!

Ready. Steady. All the best!

Project Setup

Program Launch with Flags

We will use command-line options or so-called *flags* to initialize the execution of *RescueBot*. Therefore, you need to add a few more options as possible command-line arguments.

Hence, your program is run as follows:

```
$ java RescueBot [arguments]
```

Print Help

Make sure your program provides a help documentation to tell users how to correctly call and execute your program. The help is a printout on the console telling users about each option that your program supports.

The following program calls should invoke the help:

```
$ java RescueBot --help
or
$ java RescueBot -h
```

The command-line output following the invocation of the help should look like this:

```
RescueBot - COMP90041 - Final Project

Usage: java RescueBot [arguments]

Arguments:
-s or --scenarios    Optional: path to scenario file
-h or --help        Optional: Print Help (this message) and exit
-l or --log        Optional: path to data log file
```

The help should be displayed when the *--help* or *-h* flag is set or if the *--scenarios* or *--log* flag is set without an argument (i.e., no path is provided). The flag *--scenario* or *-s* tells your program about the path and filename of the file that contains predefined scenarios. The flag *--log* or *-l* indicates the path and filename where scenarios and user judgements should be saved to. More about that later.

Multiple flags can be used and their order is interchangeable, for example:

```
$ java RescueBot -s scenarios.csv -l logfile.log
```

is equivalent to

```
$ java RescueBot -l logfile.log -s scenarios.csv
```

The help should always be displayed when your program is invoked with an invalid set of arguments.

The Class RescueBot.java

This class provided in the code skeleton holds the *main* method and manages your program execution. It needs to take care of the program parameters as described above.

This class also houses the *decide* method, which implements the decision-making algorithm.

Decision Algorithm

Your task is to implement the *public static* method *decide(Scenario scenario),* which returns an object of the type *Location,* which will be specified later. Your code must choose a location that contains a group of characters that will be saved for the given scenario.

To make the decision, your algorithm needs to consider the attributes of the characters involved as well as the situation. You can take any of the characters' characteristics (age, body type, profession, etc.) into account when making your decision, but you must base your decision on at least 5 characteristics–from the scenario itself (*e.g.*, whether the characters have illegally trespassed) or from the characters' attributes. Note that there is no right or wrong in how you design your algorithm. Execution is what matters here so make sure your code meets the technical specifications. But you may want to think about the consequences of your algorithmic design choices.

Main Menu

Once the program is launched, the main menu is shown with a welcome screen: your program should read in and display the contents of *welcome.ascii* to the user without modifying it. The message provides background information about *RescueBot* and walks the user through the program flow.

The main menu will form the core of your system and will control the overall flow of your program.

When your program is run, it should show the following output:

```
$ javac *.java
$ java RescueBot -s scenarios.csv
         __
 _(\    |@@|
(__/\__ \--/ __
   \___|----|  |   __
       \ }{ /\ )_ / _\
       /\__/\ \__O (_COMP90041
      (--/\--)    \__/
      _)(  )(_
     `___''___`

    $$$$$$$\                                                $$$$$$$\             $$\
    $$  __$$\                                               $$  __$$\            $$ |
    $$ |  $$ | $$$$$$\   $$$$$$$\  $$$$$$$\ $$\   $$\  $$$$$$\  $$ |  $$ | $$$$$$\ $$$$$$\
    $$$$$$$  |$$  __$$\ $$  _____|$$  _____|$$ |  $$ |$$  __$$\ $$$$$$$\ |$$  __$$\\_$$  _|
    $$  __$$< $$$$$$$$ |\$$$$$$\  $$ /      $$ |  $$ |$$$$$$$$ |$$  __$$\ $$ /  $$ | $$ |
    $$ |  $$ |$$   ____| \____$$\ $$ |      $$ |  $$ |$$   ____|$$ |  $$ |$$ |  $$ | $$ |$$\
    $$ |  $$ |\$$$$$$$\ $$$$$$$  |\$$$$$$$\ \$$$$$$  |\$$$$$$$\ $$$$$$$  |\$$$$$$  | \$$$$  |
    \__|  \__| \_______|\_______/  \_______| \______/  \_______|\_______/  \______/   \____/


Welcome to RescueBot!

The idea of RescueBot is based on the Trolley Dilemma, a fictional scenario presenting a decision m

The answers are not straightforward. There are a number of variables at play, which influence how p

{N} scenarios imported.
Please enter one of the following commands to continue:
- judge scenarios: [judge] or [j]
- run simulations with the in-built decision algorithm: [run] or [r]
- show audit from history: [audit] or [a]
- quit the program: [quit] or [q]
>
```

This initial output is made of two parts:

1. The welcome text, which you need from the file provided to you in the starter code
2. The number of scenarios that were imported from the *scenarios* file. {N} depicts the number. If no *scenarios* file is provided at program launch, this line should be removed.
3. An initial message explaining how the user should use the terminal, followed by a command prompt **"> "**, which waits for the user to enter a command and hit *return* to continue.

Here is where the user selects an option for the program execution. If the user enters a command that is not listed in the menu, the following message should be output, followed by the menu prompt:

```
Invalid command! Please enter one of the following commands to continue:
- judge scenarios: [judge] or [j]
- run simulations with the in-built decision algorithm: [run] or [r]
- show audit from history: [audit] or [a]
- quit the program: [quit] or [q]
>
```

Reading in Scenarios

Your program needs to support the import of scenarios from a *scenarios* file. The path to the *scenarios* file is optionally provided as a command-line argument when running your program:

```
$ java RescueBot --scenarios path/to/scenarios.csv
or
$ java RescueBot -s path/to/scenarios.csv
```

These program calls above are equivalent and should both be supported by your program.

The command-line argument following the flag *--scenarios* or *-s* respectively specifies the file path where the scenarios file (*scenarios.csv* in this case) is located. Your program should check whether the file is located at the specified location and handle a *FileNotFoundException* in case the file does not exist. In this case, your program should terminate with the following error message:

```
java.io.FileNotFoundException: could not find scenarios file.
```

If no scenarios argument is provided, your program needs to create its own scenarios randomly.

Parsing the Scenarios File

Next, your program needs to read in the scenarios file. The Table below lists the contents of the provided *scenarios.csv*, a so-called *comma-separated values* (*CSV*) file. The file contains a list of values, each separated by a comma.

	gender	age	bodyType	profession	pregnant	species	isPet
scenario:flood							
location:13.7154 N;150.9094 W;trespassing							
human	female	16	overweight	none	FALSE		
animal	female	8	overweight			cat	TRUE
animal	female	16	overweight			koala	FALSE
human	male	7	athletic	none	FALSE		
human	male	17	average	student	FALSE		
location:15.3983 N;122.8133 W;legal							
human	male	52	athletic	doctor	FALSE		
animal	male	3	overweight			koala	FALSE
scenario:bushfire							
location:17.5548 S;158.8239 E;legal							
human	male	71	athletic	none	FALSE		
animal	male	15	average			wallaby	FALSE
human	male	37	overweight	professor	FALSE		
human	female	41	average	doctor	FALSE		
location:17.6069 S;161.7064 E;trespassing							
human	female	34	average	criminal	FALSE		
human	male	59	athletic	professor	FALSE		
human	female	28	overweight	ceo	TRUE		
scenario:cyclone							
location:42.6598 S;69.283 E;legal							
human	male	61	athletic	homeless	FALSE		
animal	female	9	overweight			wallaby	FALSE
human	male	35	overweight	ceo	FALSE		
location:42.7966 S;68.3571 E;legal							
human	male	67	average	ceo	FALSE		
human	female	14	athletic	none	FALSE		
human	male	34	athletic	homeless	FALSE		

As seen in the Table, the first line contains the headers, *i.e.*, the names of each data field and can therefore be ignored by your program. Each subsequent row presents an instance of a scenario, location, or character. Scenarios are preceded by a single line that starts with *scenario:* followed by the descriptor (e.g., flood, bushfire, etc.). For each scenario, there are multiple locations, in which a group of characters are awaiting rescue. The table lists each location as *location:* followed by the location's latitude, longitude, and whether the characters are trespassing or have legally entered the location.

In the provided *scenarios.csv*, the first scenario describes a flood with characters trapped in 2 locations. The first group is trespassing and is made up of 3 humans and 2 animals. The second group has legally entered a location nearby and is made up of one human doctor and one koala.

There are two more scenarios depicted in this file, a bushfire and a cyclone event.

Your *RescueBot* class needs to be able to parse the scenarios file and create scenarios for the user judgement or simulation. Note that a scenarios file can contain any number of scenarios with any number of locations and characters. You can assume that all scenarios files follow the same column order as shown in the Table.

You will be provided with several scenarios files for testing purposes.

Handling Invalid Data Rows

While reading in the scenarios file line by line your program may encounter three types of exceptions, which your program should be able to handle:

1. Invalid Data Format

You need to handle exceptional cases where there is an invalid number of data fields per row: in case the number of values in one row is less than or exceeds 8 values an **InvalidDataFormatException** should be thrown. Your program should handle such exceptions by issuing the following warning statement to the command line, skip the respective row, and continue reading in the next line.

```
WARNING: invalid data format in scenarios file in line {X}
```

2. Invalid Number Format

This refers to an invalid data type in a cell: in case the value can not be cast into an existing data type (*e.g.*, a character where an int should be for age) a **NumberFormatException** should be thrown. More about valid characteristics later.

Your program should handle such exceptions by issuing the following warning statement to the command line, assign a default value instead, and continue with the next value in that line.

```
WARNING: invalid number format in scenarios file in line {X}
```

3. Invalid Field Values

In case your program does not accommodate a specific value (*e.g.*, *skinny* as a bodyType) an **InvalidCharacteristicException** should be thrown. Your program should handle such exceptions by issuing the following warning statement to the command line, assign a default value instead, and continue with the next value in that line.

```
WARNING: invalid characteristic in scenarios file in line {X}
```

Note that {X} depicts the line number in the scenarios file where the error was found.

The warnings above should be printed to the console in the order they are encountered after the welcome message as shown below:

```
        __
 _(\    |@@|
(__/\__ \--/ __
   \___|----|  |   __
       \ }{ /\ )_ / _\
       /\__/\ \__O (_COMP90041
      (--/\--)    \__/
      _)(  )(_
     `---''---`

    $$$$$$$\                                                  $$$$$$$\             $$\
    $$  __$$\                                                 $$  __$$\            $$ |
    $$ |  $$ | $$$$$$\   $$$$$$$\  $$$$$$$\ $$\   $$\  $$$$$$\  $$ |  $$ | $$$$$$\ $$$$$$\
    $$$$$$$  |$$  __$$\ $$  _____|$$  _____|$$ |  $$ |$$  __$$\ $$$$$$$\ |$$  __$$\\_$$  _|
    $$  __$$< $$$$$$$$ |\$$$$$$\  $$ /      $$ |  $$ |$$$$$$$$ |$$  __$$\ $$ /  $$ | $$ |
    $$ |  $$ |$$   ____| \____$$\ $$ |      $$ |  $$ |$$   ____|$$ |  $$ |$$ |  $$ | $$ |$$\
    $$ |  $$ |\$$$$$$$\ $$$$$$$  |\$$$$$$$\ \$$$$$$  |\$$$$$$$\ $$$$$$$  |\$$$$$$  | \$$$$  |
    \__|  \__| \_______|\_______/  \_______| \______/  \_______|\_______/  \______/   \____/


Welcome to RescueBot!

The idea of RescueBot is based on the Trolley Dilemma, a fictional scenario presenting a decision m

The answers are not straightforward. There are a number of variables at play, which influence how p

WARNING: invalid characteristic in scenarios file in line 7
3 scenarios imported.
Please enter one of the following commands to continue:
- judge scenarios: [judge] or [j]
- run simulations with the in-built decision algorithm: [run] or [r]
- show audit from history: [audit] or [a]
- quit the program: [quit] or [q]
>
```

Judging Scenarios

When a user selects [judge] or [j] from the main menu, the following program sequence should execute:

1. **Collect user consent** for saving data
2. **Present scenarios** from scenarios file or randomly generated
3. **Show statistic**
4. **Save judged scenarios** and decisions (if permissible)
5. **Repeat or return**: show more scenarios or return to the main menu

1. Collect User Consent

First, your program should collect the user's consent before saving any results. Explicit consent is crucial to make sure users are aware of any type of data collection. Your program should, therefore, ask for explicit user consent before logging any user responses to a file. Before showing the first scenario, your program should, therefore, prompt the user with the following question on the command line:

```
Do you consent to have your decisions saved to a file? (yes/no)
>
```

Only if the user confirms (*yes*), your program should save both the scenario and the user's judgement to the log file (if the filename is not explicitly set by the command-line flag (*--log*) save the results to the file *rescuebot.log* in the default folder. If the user selects *no*, your program should function normally but not write any of the users' decisions to the file (it should still display the statistic on the command line though). If the user types in anything other than *yes* or *no*, an **InvalidInputException** should be thrown and the user should be prompted again by displaying:

```
Invalid response! Do you consent to have your decisions saved to a file? (yes/no)
>
```

2. Present Scenarios

Once the user consented (or not), the scenario judging begins. Therefore, scenarios are either imported from the scenarios file or (if the scenarios file is not specified) randomly generated.

A scenario contains all relevant information about the type of scenario, the locations, and the characters therein. Here is an example:

```
====================================
# Scenario: flood
```

```
====================================
[1] Location: 13.7154 N, 150.9094 W
Trespassing: yes
5 Characters:
- overweight child female
- cat is pet
- koala
- athletic child male
- average adult student male
[2] Location: 15.3983 N, 122.8133 W
Trespassing: no
2 Characters:
- athletic adult doctor male
- koala
To which location should RescueBot be deployed?
>
```

Each location is numbered in ascending order. The user picks the location where RescueBot should be deployed by entering the corresponding number, which results in the saving of the characters in that location.

Characters' attributes are written in lowercase and separated by *single space*. Your output *must match* the output specifications as described below:

Character Attributes

Characters in a scenario are either humans or animals. They share some traits and differ in others. The following are some shared attributes you need to take into account:

- each character has an **age**, which should be treated as a *class invariant* for which the following statement always yields true: age >= 0.
- **gender**: must at least include the values *FEMALE* and *MALE* as well as a default option *UNKNOWN*, but can also cover more diverse options if you so choose.
- **bodyType**: must be made up of the values *AVERAGE*, *ATHLETIC*, and *OVERWEIGHT* as well as a default option *UNSPECIFIED*.

Humans

More specifically, scenarios are inhabited by humans who exhibit the following characteristics:

- **ageCategory**: depending on the age, each human should be categorized into one of the following:
  - *BABY*: a human with an age between 0 and 4.
  - *CHILD*: a human with an age between 5 and 16.
  - *ADULT*: a human with an age between 17 and 68.
  - *SENIOR*: a human with an age above 68.
- humans tend to have a **profession.** In these scenarios, you should include the following values:

- *DOCTOR*
- *CEO*
- *CRIMINAL*
- *HOMELESS*
- *UNEMPLOYED*
- *NONE* as default
- You can add your own professions as well.

Note that only *ADULTs* have professions, other age categories should be classified as *NONE*. Additionally, you are tasked with coming up with at least two more profession categories you deem feasible.

Furthermore, *female* humans can be **pregnant**. Think of it as another *class invariant* where *males* cannot be pregnant and only *ADULT females* should be able to.

Humans have a specific output format when printed to the command line. Regardless of whether you add any custom characteristics or not, in a given scenario, humans must be described as follows:

```
<bodyType> <age category> [profession] <gender> [pregnant]
```

Note that attributes in brackets [] should only be shown if they apply, \textit{e.g.}, a baby does not have a profession so, therefore, the profession is not displayed.

Here is an example:

```
athletic adult doctor female
```

or

```
average adult doctor female pregnant
```

Note that words are in lowercase and separated by single *spaces*. Age, other than age category, is ignored in the output.

Animals

At last, animals are part of the environment we live in. People walk their pets so make sure your program accounts for these. Animals share some characteristics with humans, but also have the following specific attributes:

- **species**: this indicates what type of species the animal represents, for example, a koala or dog. Make sure your program accounts for arbitrary animal species. You can also invent your own.
- **isPet**: animals can be pets. If so, it needs to be mentioned in the scenario.

Animals also have a specific output format when printed to the command line, which should follow the following specification:

```
<species> [is pet]
```

Note that only dogs, cats, and ferrets can be pets but don't need to be.

Here is a concrete example:

```
cat is pet
```

Here is another example where the animal is not a pet:

```
platypus
```

Note that words are in lowercase, separated by single *spaces*, and that *age*, *gender*, and *bodyType* are ignored in the animal's output.

Scenario Attributes

Scenarios list the different locations, in which groups of humans and animals gather. Each location has a latitude, longitude, and the information whether the characters have trespassed or legally entered the area. When printed to the command line a scenario should follow this format:

```
====================================
# Scenario: <description>
====================================
[N] Location: <latitude>, <longitude>
Trespassing: {yes, no}
{# of characters} Characters:
- <character as described above>
.
.
.
[N+1] Location: <latitude>, <longitude>
Trespassing: {yes, no}
{# of characters} Characters:
- <character as described above>
.
.
.
```

Your output *must match* these output specifications.

For user judgement, scenarios are presented on the command line one by one as described. Each scenario should be followed by the following prompt:

```
To which location should RescueBot be deployed?
>
```

If the user enters anything but a valid location number, the following message should be displayed before awaiting a new user response:

```
Invalid response! To which location should RescueBot be deployed?
>
```

After the user made a decision, the next scenario is shown, followed by yet another prompt to judge the scenario. This procedure should repeat until 3 scenarios have been shown and judged. After the third scenario decision, the result statistic is presented.

Random Scenario Generation

If no scenario file is provided when your program is executed, you need to generate them randomly. To guarantee a balanced set of scenarios, it is crucial to randomize as many elements as possible, including the number and characteristics of humans and animals involved in each scenario as well as the locations (*i.e.*, valid latitudes, longitudes, and whether groups have been trespassing).

The minimum number of locations for each scenario should be 2. No maximum needed. At least one human or animal should be present in each location.

3. Show Statistic

The statistic shows the traits of the characters RescueBot has saved. The statistic is accumulative, *i.e.*, it takes into account all scenarios that have been judged so far (not just the 3 previous ones). It should list a number of factors, including:

- age category
- gender
- body type
- profession
- pregnancy
- whether it's humans or animals
- species
- pets
- trespassing or legal access

Your statistic should account for each value of each respective characteristic, that is present in the given scenarios. For example, if you had scenarios with overweight body types, *overweight* must be listed in the statistic. If none of your scenarios included this particular body type, it must not be listed there. Also, make sure that you only update the statistic for, let's say *cats*, if a cat was present in the tested scenario. If there is no cat in a given scenario, you must not change the number of cats that survived in your statistic. Here is an example of how the statistic for cats is calculated:

$$\frac{C}{N_c}$$

where *C* is the number of rescued cats and *N_c* is the total number of cats that were present in scenarios. Here is another example for body types:

$$\frac{B}{N_b}$$

where *B* represents the number of humans with body type *b* and *N_b* represents the total number of body types *b* occurring in all the scenarios. You will need to construct the corresponding formulas for the remaining characteristics yourself. The example output below will give you some further hints.

The default values *unknown* (gender), *unspecified* (body type), and *none* (profession) should not be listed in the statistic. Further, the following characteristics should only make it into the statistic if they are related to a human:

- age category
- gender
- body type
- profession
- pregnancy

Animals are not represented in the statistic of these characteristics. Animals should only be counted for the following:

- class type (human or animal)
- species
- pets
- trespassing

This is the *output format* (with pseudocode) of the statistic:

```
====================================
# Statistic
====================================
- % SAVED AFTER <# of runs> RUNS
<for each characterstic:>
    <characterstic>: <survival ratio>
--
average age: <average>
```

Here is an example output for running the *scenarios.csv* as provided (note that when running your program, users may make different decisions, which thus leads to a different statistic):

```
====================================
# Statistic
====================================
- % SAVED AFTER 3 RUNS
cat: 1.00
child: 1.00
pet: 1.00
senior: 1.00
student: 1.00
```

```
average: 0.75
trespassing: 0.63
animal: 0.60
female: 0.60
human: 0.60
male: 0.60
legal: 0.59
athletic: 0.58
doctor: 0.50
homeless: 0.50
koala: 0.50
overweight: 0.50
professor: 0.50
wallaby: 0.50
adult: 0.46
ceo: 0.34
criminal: 0.00
pregnant: 0.00
--
average age: 33.78
```

The list of characteristics must be sorted in descending order of the survival ratio. All ratios are displayed with two digits after the decimal place (*round up* to the second decimal). If there is a tie, make sure to continue sorting in alphabetical order. Note that the last two lines are not part of the sorted statistic but are at a fixed position in the output. The average age is calculated across all *human survivors* and rounded in the same way.

If the user runs multiple scenario blocks (of 3 each), make sure to update your statistic rather than overwrite it.

4. Save Judged Scenarios

If the user previously consented to the data collection, you should save each scenario and the user's decision to the logfile. If the file path and name was explicitly provided when your program was executed (*--log),* the data should be appended (not overwritten). If no path was specified, your program should write to *rescuebot.log* in the default folder. It is up to you how to design the format of the logfile but it must be saved in ASCII code, *i.e.,* human-readable. If the file does not exist, your program should create it. If the directory specified by the file path variable does not exist, your program should print the following error message to the command line and terminate:

```
ERROR: could not print results. Target directory does not exist.
```

The logfile will keep a history of all scenarios and judgements made. It will form the basis for the audit.

5. Repeat or Return

Following the statistic, the user should be prompted to either continue with a new set of scenarios or return to the main menu:

```
Would you like to continue? (yes/no)
>
```

Should the user choose *no,* the program returns to the main menu. The judged scenarios should be reset, meaning that, should the user return to judging scenarios, a new statistic will be created. The saved file history (if consented) will continue, however.

If the user decides to continue (*yes*), the next three scenarios (or if less than three are left, the remaining ones) should be shown. If at any point, the config file does not contain any more scenarios, the final statistic should be shown followed by an exit prompt as shown below:

```
====================================
# Statistic
====================================
- % SAVED AFTER 3 RUNS
cat: 1.00
child: 1.00
pet: 1.00
senior: 1.00
student: 1.00
average: 0.75
trespassing: 0.63
animal: 0.60
female: 0.60
human: 0.60
male: 0.60
legal: 0.59
athletic: 0.58
doctor: 0.50
homeless: 0.50
koala: 0.50
overweight: 0.50
professor: 0.50
wallaby: 0.50
adult: 0.46
ceo: 0.34
criminal: 0.00
pregnant: 0.00
--
average age: 33.78
That's all. Press Enter to return to main menu.
>
```

Run Simulation

The simulation basically runs scenarios through your decision algorithm. If a scenarios file is provided, each scenario therein contained should be evaluated, and one comprehensive statistic should be shown. The statistic follows the same structures as described earlier and subsequently prompts users to return to the main menu, for example:

```
====================================
# Statistic
====================================
- % SAVED AFTER 12 RUNS
criminal: 1.00
platypus: 1.00
dingo: 0.67
snake: 0.60
pet: 0.56
student: 0.56
cockatoo: 0.50
dog: 0.50
ferret: 0.50
child: 0.45
animal: 0.40
male: 0.40
athletic: 0.39
trespassing: 0.36
professor: 0.29
adult: 0.28
human: 0.28
legal: 0.27
ceo: 0.24
overweight: 0.22
pregnant: 0.22
senior: 0.20
female: 0.18
doctor: 0.17
homeless: 0.17
koala: 0.17
wallaby: 0.17
average: 0.12
unemployed: 0.09
baby: 0.00
cat: 0.00
possum: 0.00
--
average age: 36.30
That's all. Press Enter to return to main menu.
>
```

If no scenarios file is provided, the following prompt should be shown to request the number of

scenarios that should be created for the simulation:

```
How many scenarios should be run?
>
```

If the user provides anything other than an integer, your program should throw an appropriate exception and prompt the user again until a valid number is provided:

```
Invalid input! How many scenarios should be run?
>
```

If a valid number *N* is provided, your program should create *N* random scenarios and run each through your decision algorithm. The resulting statistic should be printed to the console as above.

Scenarios and decisions of all simulations should *always* be written to the logfile. Make sure that the data structure in your log indicates whether the saved decision was made by your *algorithm* or by a *user*.

Audit from History

An audit is an inspection of your algorithm or decisions with the goal of revealing inherent biases that may be built-in as an (un)intended consequence. In this task, you will need to read all scenarios and decisions from the entire history of your logfile and create one comprehensive statistic for it.

If no logfile was specified at the start of your program, check whether the default *rescuebot.log* exists. If so, use it for the audit.

If no logfile is specified or found (or if the file is empty), throw an appropriate exception and print the below message to the console before returning to the main menu:

```
No history found. Press enter to return to main menu.
>
```

Once your program manages to read in your logfile, you need to recreate two final statistics as described earlier: one for any algorithm-based decisions and one for all user-made judgements. If either has no entries no statistic for that type is shown. Here is an example of how both statistics may look like:

```
====================================
# Algorithm Audit
====================================
- % SAVED AFTER 27 RUNS
snake: 0.56
dingo: 0.40
pregnant: 0.36
doctor: 0.34
platypus: 0.34
average: 0.32
female: 0.31
professor: 0.30
senior: 0.30
unemployed: 0.30
koala: 0.29
baby: 0.28
student: 0.28
homeless: 0.27
trespassing: 0.27
human: 0.25
adult: 0.24
child: 0.24
animal: 0.21
legal: 0.21
overweight: 0.21
athletic: 0.20
ceo: 0.18
male: 0.18
```

```
dog: 0.17
cat: 0.13
ferret: 0.12
pet: 0.10
criminal: 0.05
cockatoo: 0.00
kangaroo: 0.00
possum: 0.00
wallaby: 0.00
--
average age: 40.87

====================================
# User Audit
====================================
- % SAVED AFTER 3 RUNS
homeless: 1.00
kangaroo: 1.00
koala: 0.67
athletic: 0.50
student: 0.50
legal: 0.45
pregnant: 0.40
animal: 0.38
male: 0.38
adult: 0.34
human: 0.34
senior: 0.34
unemployed: 0.34
female: 0.30
average: 0.29
overweight: 0.29
cat: 0.00
ceo: 0.00
criminal: 0.00
dingo: 0.00
doctor: 0.00
pet: 0.00
professor: 0.00
trespassing: 0.00
wallaby: 0.00
--
average age: 47.34
That's all. Press Enter to return to main menu.
>
```

Notice the two names of each audit (*Algorithm* or *User)* and the empty line between the two statistics.

And that's it. Almost.

Documentation

Always make sure to document your code in general. For this project you need to provide two types of documentation for your program: a UML diagram depicting your overall architecture and make sure to use JavaDoc syntax so that you can create an automatic Java code documentation in HTML. Your UML diagram needs to be submitted as a PDF with the filename *architecture.pdf* along with your code. You do not need to submit your documentation created through JavaDoc. This will be created automatically.

UML Diagram

Prepare a UML diagram describing your entire program containing all classes, their attributes (including modifiers), methods, associations, and dependencies. For each class, you need to identify all its instance variables and methods (including modifiers) along with their corresponding data types and a list of parameters. You should also identify relationships between classes, including associations, multiplicity, and dependencies. Static classes must be included in the UML. You can leave out any helper function that you added (i.e., minor classes that don't play a major role other than supporting others).

Make sure to save your UML diagram in PDF format and add it to your code in the root folder as *architecture.pdf*.

Javadoc

Make sure all your classes indicate their author and general description. For each constructor and method specified in the final project description, you must provide at least the following tags:

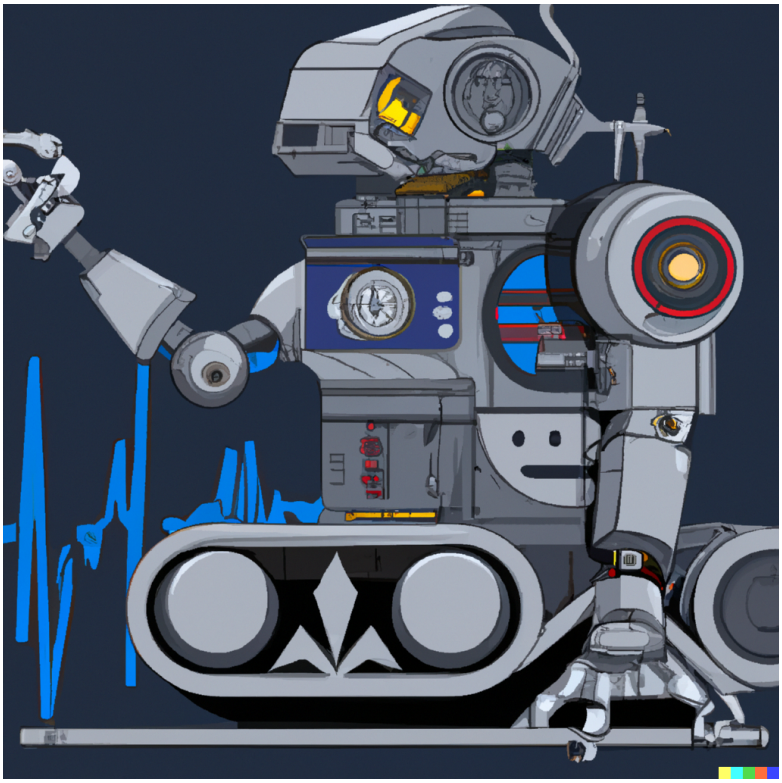
- @param tags
- @returns tag
- @throws

You can leave out minor helper functions that you may have added. Make sure to test the correct generation of your documentation using javadoc.

For the submission, you do not need to generate and add the javadoc to edstem. For marking, we will create the javadoc programmatically from your code.

RescueBot

So here we are. Make sure to read the specification documents thoroughly. Then read them again. And again.



Some starter code has been provided for you. Write the code to implement your program there

We also provide some basic tests for your code, which will run automatically every time you hit "Mark". You can run these tests as many times as you would like until the submission deadline.

⚠ Submission will close on 16 June, 11:59 pm **AEST**.

Certification of Individual Work

Note well that this project is your final assessment so cheating will not be tolerated. You are not to use any AI tools for this assignment. Further, any form of material exchange, whether written, electronic or through any other medium, is considered cheating, and so is the soliciting of help from electronic newsgroups. Providing undue assistance is considered as serious as receiving it, and in the case of similarities that indicate an exchange of more than basic ideas, formal disciplinary action will be taken for all involved parties. A sophisticated program that undertakes a deep structural analysis of Java code identifying regions of similarity will be run over all submissions in "*compare every pair*" mode (yes, it catches on if lines are moved around or variable names changed. It also detects code generation programs, such as Chat-GPT). Beating these systems takes at least as much or more effort than doing the project by yourself.

By submitting your work on edstem you certify that the work submitted was done independently and without unauthorized aid.

In the name of the entire teaching team, we wish you all the best for your submission and sincerely hope you enjoy working on this coding challenge. You have all come a long way!

Reflection (Bonus Task)

Question

This optional task gives you the opportunity to reflect on and describe the reasons you applied when designing your program's decision-making as well as the consequences revealed by auditing your algorithm. For example, what were inherent biases you might have become aware of by running this simulation? Were there any surprises? What are the consequences of design choices that you take as a programmer in general? Please make sure to stay below 250 words.

This part is optional but allows you to score an additional 2 points. Note that you cannot exceed the total of 40 points for the entire project, but we would love to read about your thoughts.

WARNING: you can only submit your reflection once, so choose your words wisely :)

No response

Assessment

The final project is worth 40% of the total marks for the subject. Your Java program will be assessed based on the correctness of the output as well as the quality of code implementation.

We will use automated tests to compile, run, and compare your outputs for several test cases with generated expected outputs. The automatic test will deem your output wrong if your output does not match the expected output, even if the difference is just having extra space or missing a colon. Therefore, it is crucial that your output follows exactly the same format shown in the provided examples.

Passing the tests we provide here on edstem does not mean you will score full marks. Each submission will further be subject to hidden tests and, more importantly, manual inspection by human markers.

But, more importantly, we expect you to apply good coding practices and make use of appropriate concepts as discussed throughout the lecture. A general marking rubric will be released here on edstem.

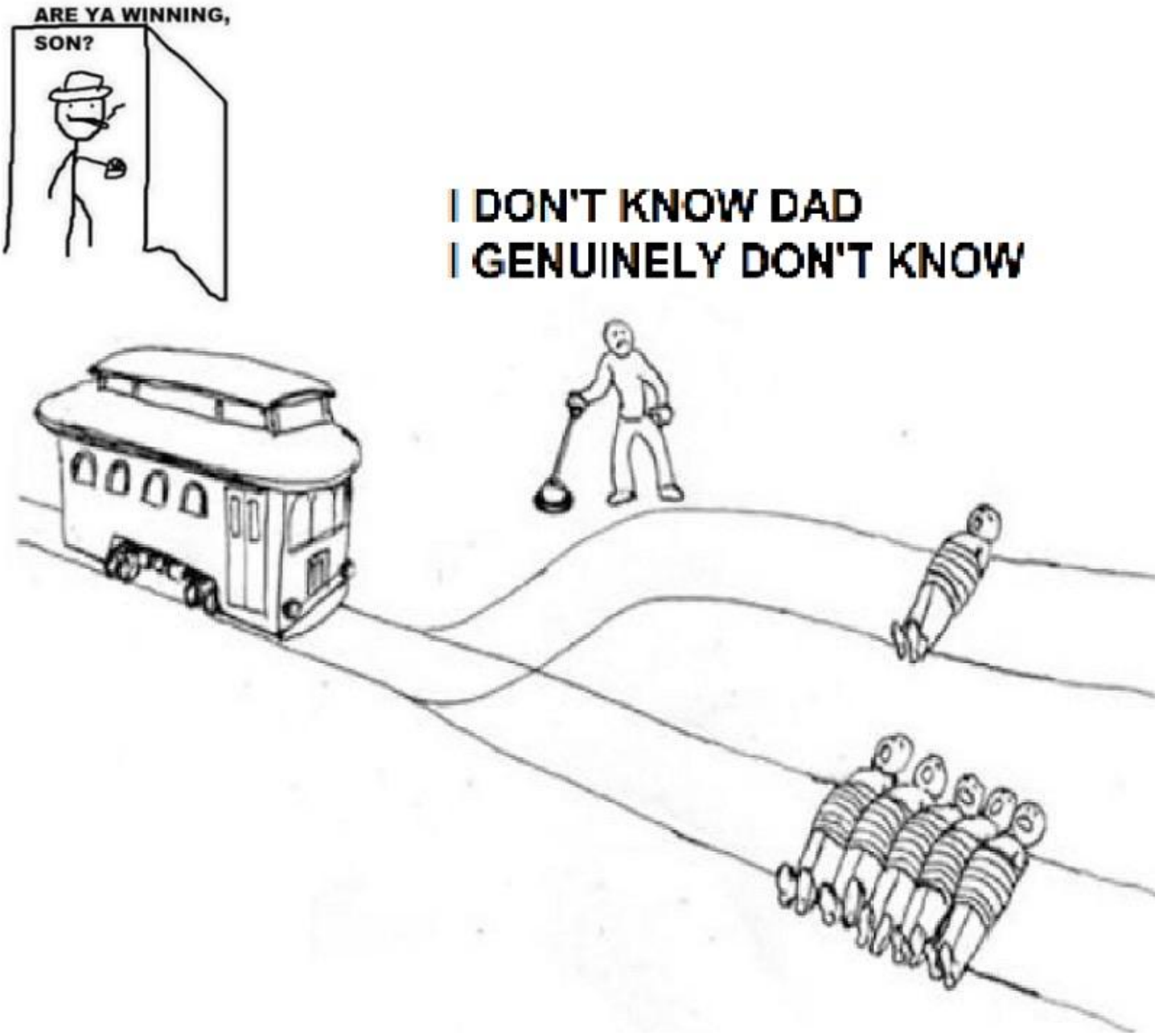
Your submission is only valid if your code compiles and runs on edstem! Arguments, such as *"It runs on my local machine"*, can and will not be accepted.

What will be graded?

We will only mark the **last** version of your program committed before the submission deadline. So make sure you submit code to edstem early and regularly.

Be aware of timezone differences! Submissions via email or Canvas will not be accepted.

BUT we believe in you! And are looking forward to your unique and ingenious solutions to this fascinating problem!


ARE YA WINNING, SON?
I DON'T KNOW DAD
I GENUINELY DON'T KNOW

Marking Scheme

COMP90041 Final Project: Marking Scheme

Student: ***[Student ID goes here]***

Marking process:

1. Code compiles (Pre-requisite)
2. Run tests
3. Verify test results
4. Inspect code and fill in the sections below

1. Program Execution: Automated Tests

Visible tests passed: /5

Hidden tests passed: /10

Total tests passed: /15

```
Points awarded for this section: /15
```

2. System Architecture

Including: Code structure resembling real-world entities. Clear hierarchy by using separate, well-named files for generics, abstract, and concrete classes. Use of package(s) to modularise code.

```
Points awarded for this section: /2
```

3. OOP & Encapsulation

Including: Proficient use of modifiers (private, public) with appropriate getter/setter methods. Considering privacy leaks. Well-considered usage of static methods and free of redundant object passings.

```
Points awarded for this section: /5
```

4. Polymorphism

Including: Proficient use of generics, abstract classes or interfaces, and inheritance. Classes are well-designed without redundancies and with elegant overloading.

```
Points awarded for this section: /3
```

5. Control Flow

Including: Easily traceable program flow. Loops have clear breakout conditions. Proficient use of switch and if-else statements.

```
Points awarded for this section: /3
```

6. Data Structures & Algorithms

Including: Proficient use of a wide range of data structures. Elegant algorithmic design. Proficient use of the Java standard library and of java data types.

```
Points awarded for this section: /2
```

7. Scenario Randomization

Including: True, well-balanced randomisation of all features and characteristics.

```
Points awarded for this section: /2
```

8. Logfile

Including: Easy to read in ASCII format. Contains all information necessary to reproduce the scenario and decisions without redundancies.

```
Points awarded for this section: /2
```

9. Style

Including: Consistency around naming conventions and descriptive naming of classes, methods, and variables. Program code is well indented, spaces are sufficient, and avoidance of overly long lines of code.

```
Points awarded for this section: /2
```

10. Documentation and Javadoc

Run

```
javadoc -d doc *.java
```

Including: Javadoc created without errors. Following javadoc conventions. All major classes and methods are annotated. Additional in-line comments clarify complex program sections.

```
Points awarded for this section: /2
```

11. UML

Including: Major classes and methods are listed with their correct association arrows and multiplicity values. The spatial arrangement allows for easy reading and shows a clear hierarchy between classes. Package allocation is correctly displayed.

```
Points awarded for this section: /2
```

12. Bonus Task: Reflection

This part is optional but allows students to score an additional 2 points. Note that the points cannot exceed the total of 40 points for the entire project, including the oral exam.

Including: Logical, honest, and rigorous. Reflection shows a critical engagement with the work and identifies alternative paths forward without being vague. Student clearly explains the reasoning behind the decision engine student, articulates how these choices affected the outcome student, and critically reflects on these design choices as well as identifies one or more implications of their reflection on future coding ethics.

```
Points awarded for this section: /2
```

Total Marks for the final project: / 40

Overall comments from marker: ***[Comments go here]***

Assignment Marker: ***[Assignment marker name goes here]***

If you have any questions regarding your mark, please contact the lecturers