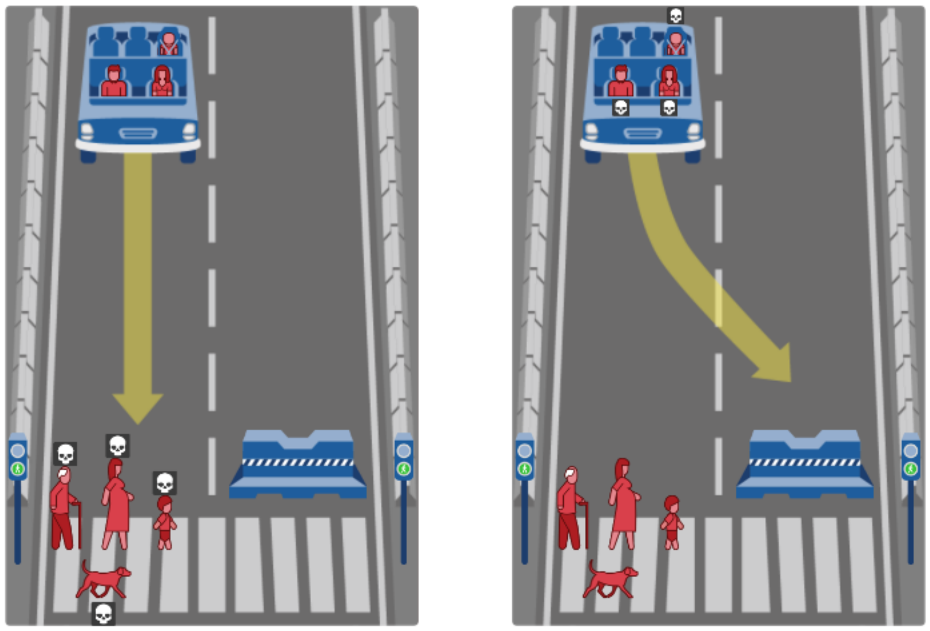
Java Project

▼ **Introduction**

The idea of Ethical Engines is based on the **Trolley Dilemma**, a fictional scenario presenting a decision-maker with a moral dilemma: choosing "the lesser of two evils". The scenario entails an autonomous car whose brakes fail at a pedestrian crossing. As it is too late to relinquish control to the car's passengers, the car needs to make a decision based on the facts available about the situation. The Figure below shows an example scenario.

In this project, you will create an **Ethical Engine**, a program designed to explore different scenarios, build an algorithm to decide between the life of the car's passengers vs. the life of the pedestrians, audit your decision-making algorithm through simulations, and allow users of your program to judge the outcomes themselves.



Scenario example above: a self-driving car approaches a pedestrian crossing but its breaks fail. Your algorithm needs to decide between two cases. Left: The car will continue ahead and drive through the crossing resulting in one elderly man, one pregnant woman, one boy, and one dog losing their lives. Right: The car will swerve and crash into a concrete barrier resulting in the death of its passengers: one woman, one man, and one baby. Note that the pedestrians abide by the law as they are crossing on a green signal (image source: http://moralmachine.mit.edu/).

▼ **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 (**EthicalEngine**), 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.

▼ **Project Setup:**

Program Launch with Flags

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

Hence, your program is run as follows:

```
$ java EthicalEngine [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 EthicalEngine --help
or
$ java EthicalEngine -h
```

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

```
EthicalEngine - COMP90041 - Final Project

Usage: java EthicalEngine [arguments]

Arguments:
    -c or --config        Optional: path to config 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 `--config` or `--log` flag is set without an argument (i.e., no path is provided).

The flag `--config` or `-c` tells your program about the path and filename of the config file that contains predefined scenarios. The flat `--log` or `-l` indicates the path and filename where scenarios and user judgements should be saved to. More about that later.

The Class EthicalEngine.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 a value of the Enumeration type *Decision,* which is either *PEDESTRIANS* or *PASSENGERS*. Your code must choose whom to save for any 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, pets, 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 it's a legal crossing) 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 *Ethical Engines* 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 EthicalEngine -c config.csv
                       __-------__
                     / _---------_ \
                    / /           \ \
                    | |           | |
                    |_|___________|_|
                /-\|                 |/-\
               | _ |\       0       /| _ |
               |(_)| \      !      / |(_)|
               |___|__\_____!_____/__|___|
               [_______|COMP90041|_______]
                ||||    ~~~~~~~~~    ||||
                `--'                 `--'


    $$$$$$$$\ $$\     $$\       $$\                    $$\
    $$  _____|$$ |    $$ |      \__|                   $$ |
    $$ |    $$$$$$\   $$$$$$$\  $$\  $$$$$$$\ $$$$$$\  $$ |
    $$$$$\  \_$$  _|  $$  __$$\ $$ |$$  _____|\____$$\ $$ |
    $$  __|   $$ |    $$ |  $$ |$$ |$$ /      $$$$$$$ |$$ |
    $$ |      $$ |$$\ $$ |  $$ |$$ |$$ |     $$  __$$ |$$ |
    $$$$$$$$\ \$$$$  |$$ |  $$ |$$ |\$$$$$$$\\$$$$$$$ |$$ |
    \________| \____/ \__|  \__|\__| \_______|\_______|\__|


    $$$$$$$$\                     $$\
    $$  _____|                    \__|
    $$ |      $$$$$$$\   $$$$$$\  $$\ $$$$$$$\   $$$$$$\   $$$$$$$\
    $$$$$\    $$  __$$\ $$  __$$\ $$ |$$  __$$\ $$  __$$\ $$  _____|
    $$  __|   $$ |  $$ |$$ /  $$ |$$ |$$ |  $$ |$$$$$$$$ |\$$$$$$\
    $$ |      $$ |  $$ |$$ |  $$ |$$ |$$ |  $$ |$$   ____| \____$$\
    $$$$$$$$\ $$ |  $$ |\$$$$$$$ |$$ |$$ |  $$ |\$$$$$$$\ $$$$$$$  |
    \________|\__|  \__| \____$$ |\__|\__|  \__| \_______|\_______/
                        $$\   $$ |
                        \$$$$$$  |
                         \______/


Welcome to Ethical Engines!

The idea of Ethical Engines is based on the Trolley Dilemma, a fictional scenario presenting a decision-maker with a moral dilemma: cho

The answers are not straightforward. There are a number of variables at play, which influence how people may feel about the decision: t

{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 *config* file. {N} depicts the number. If no *config* 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.

▼ **Reading in the Config File:**

Reading in the Config File:

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

```
$ java EthicalEngine --config path/to/config.csv
or
```

```
$ java EthicalEngine -c path/to/config.csv
```

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

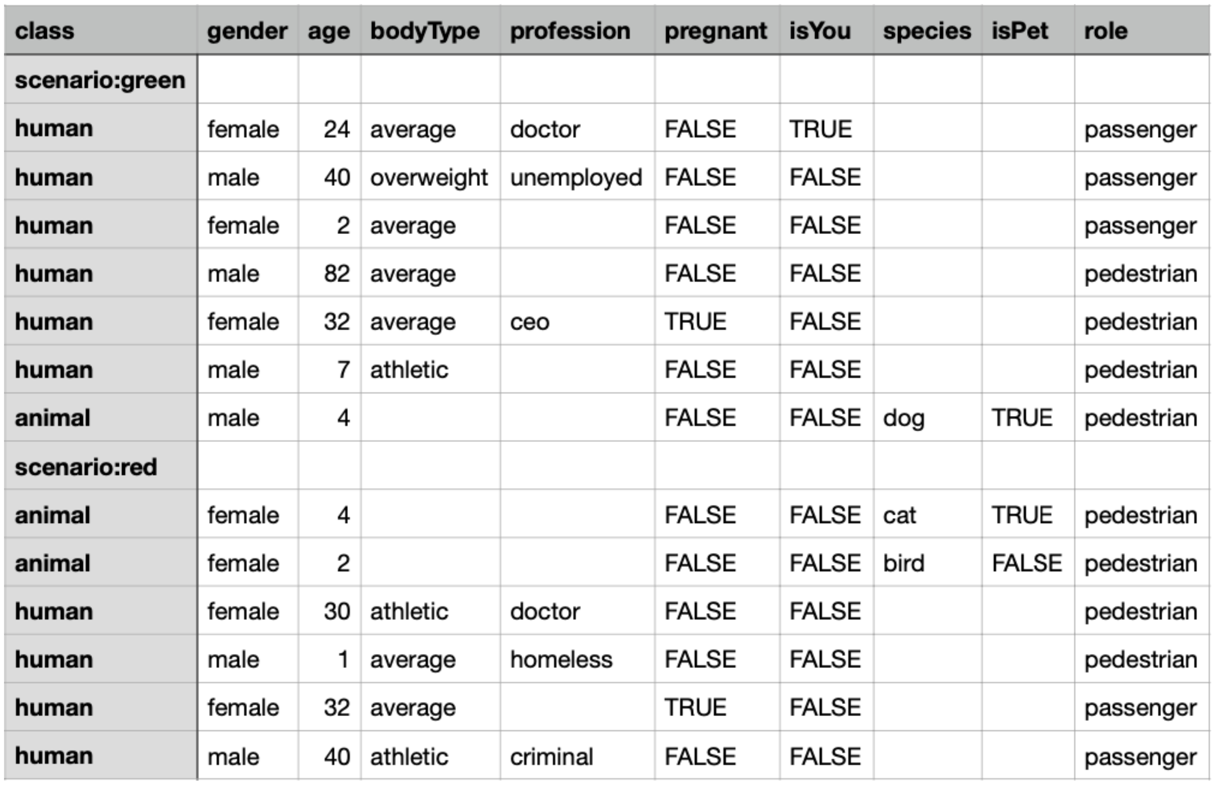
The command-line argument following the flag `--config` or `-c` respectively specifies the file path where the configuration file (*config.csv* in this case) is located. Your program should check whether the file is located at the specified location and handle a `FileNotFoundException` in case the file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

ERROR: could not find config file.

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

Parsing the Configuration File

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



class	gender	age	bodyType	profession	pregnant	isYou	species	isPet	role
scenario:green									
human	female	24	average	doctor	FALSE	TRUE			passenger
human	male	40	overweight	unemployed	FALSE	FALSE			passenger
human	female	2	average		FALSE	FALSE			passenger
human	male	82	average		FALSE	FALSE			pedestrian
human	female	32	average	ceo	TRUE	FALSE			pedestrian
human	male	7	athletic		FALSE	FALSE			pedestrian
animal	male	4			FALSE	FALSE	dog	TRUE	pedestrian
scenario:red									
animal	female	4			FALSE	FALSE	cat	TRUE	pedestrian
animal	female	2			FALSE	FALSE	bird	FALSE	pedestrian
human	female	30	athletic	doctor	FALSE	FALSE			pedestrian
human	male	1	average	homeless	FALSE	FALSE			pedestrian
human	female	32	average		TRUE	FALSE			passenger
human	male	40	athletic	criminal	FALSE	FALSE			passenger

As can be seen in the Table, the first line contains the **headers**, *i.e.*, the names (and description) of each data field and can therefore be ignored by your program.

Each subsequent row presents an **instance of a character or scenario.** Scenarios are preceded by a single line that starts with *scenario:* and indicates whether the scenario depicts a legal (green) or illegal (red) crossing.

In this case, the first scenario describes a legal crossing (scenario:green) with 3 passengers and 4 pedestrians (one of which is a dog). In fact, the first data set describes the scenario depicted in the Figure from the Introduction.

The **second scenario** describes an illegal crossing with 4 pedestrians (2 animals) and 2 car passengers.

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

Handling Invalid Data Rows

While reading in the config 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 10 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 config 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. 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 config 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 config file in line {X}
```

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

▼ **Judging Scenarios:**

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 config 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 user 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 *ethicalengines.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 config file or (if the config file is not specified) randomly generated.

A scenario contains all relevant information about the car's passengers, the pedestrians on the street, and whether the pedestrians are crossing legally. Here is an example of a legal crossing:

```
======================================
# Scenario
======================================
Legal Crossing: yes
Passengers (4)
- cat is pet
- overweight child male
- average senior female
- athletic adult ceo female pregnant
Pedestrians (3)
- average baby male
- average adult doctor male
- overweight adult homeless female
```

Here is another example with *you* in the car and a (non-pregnant) woman and pedestrians crossing the street at a red light (illegal crossing):

```
======================================
# Scenario
======================================
Legal Crossing: no
Passengers (2)
- you average baby male
- average adult criminal female
Pedestrians (2)
- average senior male
- average senior female
```

Note that a character's attributes are written in lower case and separated by *single space*. Your output *must match* the output specifications as described below:

Character Attributes

Characters in a scenario are participants in the scenario. They are either pedestrians or passengers. For this project, you will only need to consider humans and 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 include 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

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.

Now, *you* can also be part of a scenario. *You* can only be human and is representative of the user of your program. In each scenario, *you* can only appear once but does not have to be part of it.

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:

```
[you] <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
```

Similarly, here is an example if the human is *you*:

```
you average baby male
```

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 don't have human-specific characteristics but have the following specific attributes:

- **species**: this indicates what type of species the animal represents, for example, a cat or dog.

- **pet**: 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]
```

Here is a concrete example:

```
cat is pet
```

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

```
bird
```

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 are inhabited by passengers and pedestrians. Additionally, scenarios should indicate whether the crossing is **legal**, i.e., pedestrians crossing a green light. When printed to the command line a scenario should follow this format:

```
======================================
# Scenario
======================================
Legal Crossing: <yes/no>
Passengers ({# of passengers})
- <character as described above>
.
.
Pedestrians ({# of pedestrians})
- <character as described above>
.
.
```

Here is an example of a legal crossing (green light):

```
======================================
# Scenario
======================================
Legal Crossing: yes
Passengers (4)
- cat is pet
- overweight child male
- average senior female
- athletic adult ceo female pregnant
Pedestrians (3)
- average baby male
- average adult doctor male
- overweight adult homeless female
```

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:

```
Who should be saved? (passenger(s) [1] or pedestrian(s) [2])
```

Any of the following user inputs should be considered saving the passengers:

- passenger

- passengers
- 1

Any of these user inputs should be considered saving the pedestrians:

- pedestrian
- pedestrians
- 2

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 config 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 scenario itself (*e.g.*, whether it's a legal crossing or not), and whether *you* is in it or not.

The minimum number of pedestrians and passengers should be 1 respectively. The maximum number can be 60 (think of semi-autonomous buses, for example).

3. Show Statistic

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
- legality (red or green light)
- pedestrian or passenger

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 cat survivors and *N_c* is the total number of cats 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
- legality (red or green light)
- pedestrian or passenger

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 *config.csv* as provided (note that when running your program, users may make different decisions, which thus leads to a different statistic):

```
======================================
# Statistic
======================================
- % SAVED AFTER 2 RUNS
animal: 1.00
bird: 1.00
cat: 1.00
ceo: 1.00
child: 1.00
dog: 1.00
pedestrians: 1.00
pet: 1.00
senior: 1.00
athletic: 0.67
red: 0.67
male: 0.60
green: 0.58
average: 0.50
baby: 0.50
doctor: 0.50
human: 0.50
pregnant: 0.50
female: 0.40
adult: 0.34
criminal: 0.00
overweight: 0.00
passengers: 0.00
unemployed: 0.00
you: 0.00
--
average age: 30.40
```

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 *ethicalengine.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
pregnant: 1.00
you: 1.00
senior: 0.67
passengers: 0.60
female: 0.50
pet: 0.50
average: 0.37
animal: 0.34
adult: 0.29
green: 0.28
human: 0.27
male: 0.24
pedestrians: 0.16
athletic: 0.00
ceo: 0.00
child: 0.00
criminal: 0.00
dog: 0.00
homeless: 0.00
overweight: 0.00
unemployed: 0.00
--
average age: 67.25
```

▼ **Run Simulation:**

Run Simulation:

The simulation basically runs scenarios through your decision algorithm. If a config 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 prompt users to return to the main menu, for example:

```
======================================
# Statistic
======================================
- % SAVED AFTER 3 RUNS
cat: 1.00
pregnant: 1.00
you: 1.00
senior: 0.67
passengers: 0.60
female: 0.50
pet: 0.50
average: 0.37
animal: 0.34
adult: 0.29
green: 0.28
human: 0.27
male: 0.24
pedestrians: 0.16
athletic: 0.00
ceo: 0.00
child: 0.00
criminal: 0.00
dog: 0.00
homeless: 0.00
overweight: 0.00
unemployed: 0.00
--
average age: 67.25
That's all. Press Enter to return to main menu.
```

If no config 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 an *algorithm* or *user*.

▼ **Audit from History:**

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 a (un)intended consequence. In this task, you will need to read in 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 *ethicalengine.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 100 RUNS
cat: 1.00
pregnant: 1.00
you: 1.00
senior: 0.67
passengers: 0.60
female: 0.50
pet: 0.50
animal: 0.34
adult: 0.29
green: 0.28
human: 0.27
male: 0.24
pedestrians: 0.16
--
average age: 33.27

======================================
# User Audit
======================================
- % SAVED AFTER 27 RUNS
passengers: 0.8
pregnant: 0.06
senior: 0.52
passengers: 0.51
female: 0.50
pet: 0.40
animal: 0.34
adult: 0.29
you: 0.28
green: 0.27
human: 0.26
male: 0.21
pedestrians: 0.11
--
average age: 56.27
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:**

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 *ethicalarchitecture.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 *ethicalarchitecture.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.