COMP90041: Final Project
Lecturers: Dr. Tilman Dingler, Dr Thuan Pham
Submission Due: 5pm (AEDT), Friday 20 Nov, 2020.

This is your final project for COMP90041. It is equivalent of a final exam and, therefore, counts 60% towards your final grade. Consequently, the total number of points you can collect by completing this final assessment is 60 points. The number of points available is listed for each Section. All the best!

Moral Machines

The idea of Moral Machines 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. Figure 1 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.

1 Build an Ethical Engine (15 points)

Go to *https://classroom.github.com/a/ZLuymdEg* and accept the final project assignment. For details on how to check out the repository, make sure to consult the Lab 3 Materials[1].

Your program should consist of seven core classes:

```
ethicalengine/
|-- Persona.java
|-- Human.java
|-- Animal.java
|-- Scenario.java
|__ ScenarioGenerator.java
Audit.java
EthicalEngine.java
welcome.ascii
```

You can create additional classes if needed. *EthicalEngine.java* contains the *main* function and coordinates the program flow. *Scenario.java* contains a list of passengers, a list of pedestrians, as well as additional scenario conditions, such as whether pedestrians are legally crossing at the traffic light. The decision-making algorithm is implemented as a *static* method with the name *decide(Scenario scenario)* in *EthicalEngine.java*. The *welcome.ascii* file contains a text message that needs to be imported by your program and printed to the console to introduce the user task (interactive mode). Start by implementing the classes *Persona.java*, *Human.java*, *Animal.java*, and *Scenario.java*. *SenarioGenerator.java* is explained in detail in Section 2 and *Audit.java* in Section 3.

[1] https://canvas.lms.unimelb.edu.au/courses/1507/assignments/138029

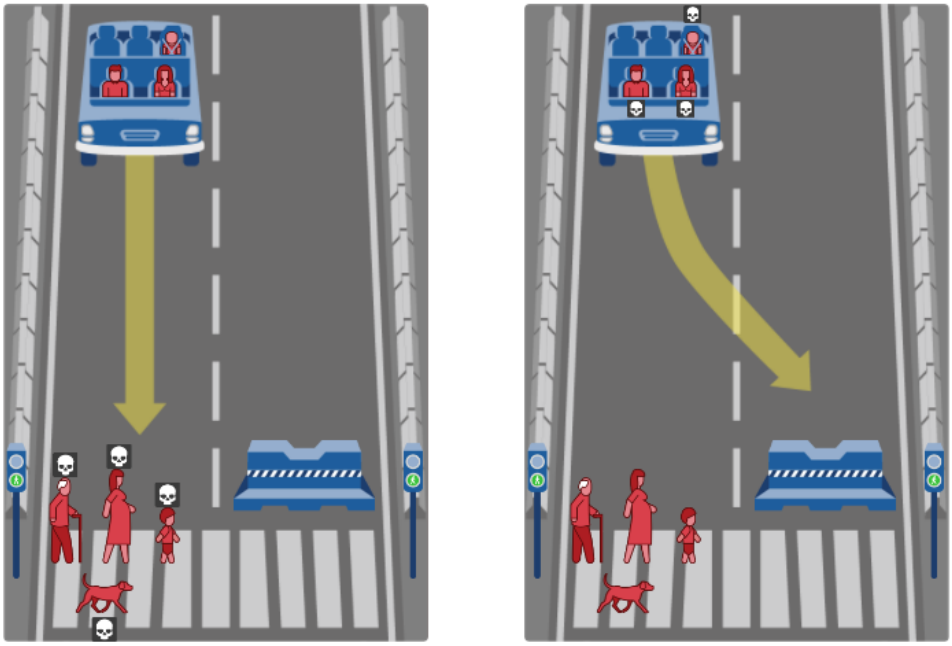

Figure 1: Scenario example: a self-driving car approaches a pedestrian crossing but its breaks fail. Your algorithm needs to decide between two cases. *Left*: The car will continue ahead and drive through the crossing resulting in one elderly man, one pregnant woman, one boy, and one dog losing their lives. *Right:* The car will swerve and crash into a concrete barrier resulting in the death of its passengers: one women, 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/).

The classes *Persona*, *Human*, *Animal*, *Scenario*, and *ScenarioGenerator* must be part of the package *ethicalengine*. The classes *Audit* and *EthicalEngine* should not be within a package.

1.1 The Abstract Class *Persona*

Persona is an *Abstract Class* from which all Persona types inherit. This base class should be implemented as depicted in Figure 2. The class further comprises two enumeration types:

1. *Gender* must include the types *FEMALE* and *MALE* as well as a default option *UNKNOWN*, but can also include more diverse options if you so choose.
2. *BodyType* includes the types *AVERAGE*, *ATHLETIC*, and *OVERWEIGHT* as well as a default option *UNSPECIFIED*.

The *Persona* Class should implement the constructors as depicted in Figure 2. Make sure the empty constructor initializes all attributes with appropriate default values.
Age should be treated as a *class invariant* for which the following statement always yields true: $age >= 0$.

1.2 Classes Inheriting from *Persona.java*

Create at least two concrete classes that directly inherit from the abstract class *Persona*:

1. *Human.java*: scenarios are inhabited by people who exhibit a number of characteristics (e.g., age, gender, body type, profession etc.). In the scenarios, each human is either considered to be a passenger or a pedestrian. Only a human can be *you*.
2. *Animal.java*: animals are part of the environment we live in. People walk their pets so make sure your program accounts for these.

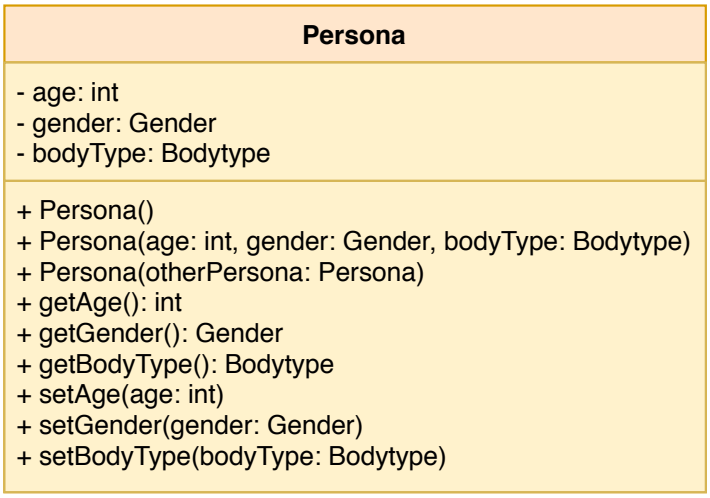

Figure 2: UML Diagram for *Persona.java*

1.2.1 The Class *Human.java*

This class represents a human in the scenarios. On top of its parent methods, the class *Human* must at least include the following public methods:

- the constructor *Human(int age, Profession profession, Gender gender, BodyType bodytype, boolean isPregnant)*.
- the copy constructor *Human(Human otherHuman)*.
- *getAgeCategory()*: returns an enumeration value of the type *AgeCategory* depending on the Human's age with one of the following values:
  - *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.
- the public method *getProfession()*: returns an enumeration value of the type *Profession*, which must include the following values: *DOCTOR*, *CEO*, *CRIMINAL*, *HOMELESS*, *UNEMPLOYED*, or *NONE* (as default). Only ADULTs have professions, other age categories should return the default value *NONE*. Additionally, you are tasked with coming up with at least two more categories you deem feasible.
- the public method *isPregnant()*: returns a boolean indicating whether the human is pregnant. For all instances of *Human* whose gender is not *FEMALE* this should return *false*.
- the public method *setPregnant(boolean pregnant)*: sets the value returned by *isPregnant() while preventing invalid states, such as a pregnant male.*
- *isYou()*: returns a boolean indicating whether the human is representative of the user, *e.g.*, you are one of the passengers in the car.
- the public method *setAsYou(boolean isYou)*: sets the value of whether the human is representative of the user.
- the public method *toString()* must output a human's characteristics according to the format shown below.

Pregnancy should be treated as a *class invariant* for which the following statement always yields true: if the human's gender is not female, the human cannot be pregnant. Also, only humans who belong to the age category *ADULT* have a profession.
The public method *toString()* must return the following **output format** when printed to the command-line:

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

Note that attributes in brackets [] should only be shown if they apply, *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 is ignored in the output.

1.2.2 The Class *Animal.java*

This class represents animals in the scenarios. On top of its parent methods, the class *Animal* must include the following public methods:

- the constructor *Animal(String species)*.
- the copy constructor *Animal(Animal otherAnimal)*.
- the public method *getSpecies()*: returns a String indicating what type of species the animal represents.
- the public method *setSpecies(String species)*: sets the value returned by *getSpecies()*.
- the public method *isPet()*: returns a boolean value depending whether the animal is a pet or wild animal.
- the public method *setPet(Boolean isPet)*: sets the value returned by *isPet()*.
- the public method *toString()* must output a pet's Personaistics according to the format shown below.

The public method *toString()* must return the following **output format** when printed to the command-line:

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

Here is an example:

```
cat is pet
```

Here is another example where *isPet()* returns *false*:

```
bird
```

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

1.3 The Class *Scenario.java*

This class contains all relevant information about a presented scenario, including the car's passengers and the pedestrians on the street as well as whether the pedestrians are crossing legally.
Each scenario can have only one instance of *Human* for which *isYou()* returns true.
The following public methods must be implemented:

- the constructor *Scenario(Persona[] passengers, Persona[] pedestrians, boolean isLegalCrossing)*: you can use Arrays or ArrayLists in your class, but you need to make sure this constructor takes a Human array as an argument.
- the public method *hasYouInCar()*: returns a boolean indicating whether you (the user) is in the car.
- the public method *hasYouInLane()*: returns a boolean indicating whether you (the user) are in the lane, *i.e.*, crossing the street.
- the public method *getPassengers()*: returns the cars' passengers as a Persona[] array.
- the public method *getPedestrians()*: returns the pedestrians as a Persona[] array.
- the public method *isLegalCrossing()*: returns whether the pedestrians are legally crossing at the traffic light.
- the public method *setLegalCrossing(boolean isLegalCrossing)*: sets whether the pedestrians are legally crossing the street.
- the public method *getPassengerCount()*: returns the number of passengers in the car (in *int*).
- the public method *getPedestrianCount()*: returns the number of pedestrians on the street (in *int*).
- the public method *toString()* must output the scenario according to the format shown below.

The public method *toString()* must return the following **output format** when printed to the command-line:

```
=====================================
# Scenario
=====================================
Legal Crossing: <yes/no>
Passengers (<getPassengerCount>)
- <Persona.toString>
.
.
Pedestrians (<getPedestrianCount>)
- <Persona.toString>
.
.
```

Here is an example for 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
```

Here is another example with you in the car and a (non-pregnant) women 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 persona's characteristics are written in lower case and separated by single *spaces*. Your output **must match** the output specifications.

1.4 The Class *EthicalEngine.java*

This class holds the *main* method and manages your program execution. It takes care of program parameters (see Section 4) as well as user input (see Section 5).
This class also houses the *decide(scenario)* method, which implements the decision-making algorithm outputting either *PEDESTRIANS* or *PASSENGERS* depending on whom to save.

Decision Algorithm Your task is to implement the *public static* method *decide(Scenario scenario)* that either 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 characteristics of the personas involved as well as the situation. You can take any of the personas' 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 personas' 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.

2 Scenario Generator (10 points)

The class *ScenarioGenerator.java* will be the basis of your simulation and shall be used to create a variety of scenarios. 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.
To be able to properly test your scenarios and make sure your results can be replicated, you must apply *pseudorandomness*. Therefore, you need to familiarize yourself first with the class *java.util.random* [2] and especially with the function *setSeed(long seed)*.
ScenarioGenerator.java must, therefore, include the following methods:

- the empty constructor *ScenarioGenerator()*: this constructor should set the seed to a truly random number

[2] https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

- the constructor *ScenarioGenerator(long seed)*: this constructor sets the seed with a predefined value
- the constructor *ScenarioGenerator(long seed, int passengerCountMinimum, int passengerCountMaximum, int pedestrianCountMinimum, int pedestrianCountMaximum)*: this constructor sets the seed as well as the minimum and maximum number for both passengers and pedestrians with predefined values
- the public method *setPassengerCountMin(int min)*: sets the minimum number of car passengers for each scenario
- the public method *setPassengerCountMax(int max)*: sets the maximum number of car passengers for each scenario
- the public method *setPedestrianCountMin(int min)*: sets the minimum number of pedestrians for each scenario
- the public method *setPedestrianCountMax(int max)*: sets the maximum number of pedestrians for each scenario
- the public method *getRandomHuman()* which returns a newly created instance of *Human* with random age, gender, bodyType, profession, and state of pregnancy
- the public method *getRandomAnimal()* which returns a newly created instance of *Animal* with random age, gender, body type, species, and whether it is a pet or not
- the public method *generate()* which returns a newly created instance of *Scenario* containing a random number of passengers and pedestrians with random characteristics as well as a randomly red or green light condition with you (the user) being either in the car, on the street, or absent.

The method *generate()* will need to abide by the minimum and maximum counts previously set for passengers and pedestrians in the scenario. If these values have not been explicitly set they need to be implicitly (*i.e.,* by default) set to 1 and 5 respectively. A minimum may never be larger than its corresponding maximum.

3 Audit your Algorithm (10 points)

An audit is an inspection of your algorithm with the goal of revealing inherent biases that may be built in as an (un)intended consequence. In this task, you will simulate a variety of scenarios and have your *EthicalEngine* decide on their outcomes.
The class *Audit.java* should:

1. create a specific number of random scenarios,
2. allow your *EthicalEngine* to decide on each outcome,
3. and summarize the results for each characteristic in a so-called *statistic of projected survival*.

The following methods must, therefore, be implemented:

- the empty constructor *Audit()*
- the public method *run(int runs)*: runs the simulation by creating $N = runs$ scenarios and running each scenario through the *EthicalEngine* using its *decide(Scenario scenario)* method. For each scenario you need to save the outcome and add the result to your statistic
- the public method *setAuditType(String name)*: sets the name of the audit type. For example: *Algorithm* for an audit of your algorithm.
- the public method *getAuditType()*: returns the name of the audit. Default should be *Unspecified*.

- the public method *toString()*: returns a summary of the simulation in the format depicted below. If no simulation has been run, this method returns "no audit available".
- the public method *printStatistic()*: prints the summary returned by the *toString()* method to the command-line.

3.1 Statistic of Projected Survival

Your statistic should list a number of factors, including:

- age category
- gender
- body type
- profession
- pregnant
- class type (human or animal)
- species
- pets
- legality (red or green light)
- pedestrian or passenger

Your statistic should account for each value of each respective characteristic, that are 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 % of cats that survived in your audit. 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 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 test outputs will give you some 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
- pregnant

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:

```
======================================
# <auditType> Audit
======================================
- % SAVED AFTER <int run> RUNS
<for each characterstic:>
    <characterstic>: <survival ratio>
--
average age: <average>
```

Here is an example output for running the *config.csv* (note that your algorithm may make different decisions and thus lead to a different statistic):

```
======================================
# Algorithm Audit
======================================
- % 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 survivors of class *Human* (animals are excluded) and rounded in the same way.

3.1.1 Update your Statistic within an Audit

If you run multiple scenarios within a particular audit, make sure to update your statistic rather than overwrite it. For example, you may run an audit subsequently over 10 (*audit.run(10)*), 50 (*audit.run(50)*), and 100 (*audit.run(100)*) scenarios and print an updated statistic after each run to the command-line. The result on the command-line should be three statistic outputs: the first with 10, the second with 60, and the last with 160 runs.

3.2 Save your Audit Results

To save the results of your audit to a file, add the public method *printToFile(String filepath)* to your *Audit* class. The method prints the results of the *toString()* method to a target file specified by the *filepath* variable. The *filepath* value (*e.g.*, 'logs/results.log') is set by the command-line flag $-r$ or $--results$ and includes both the target directory (*logs/*, in this case) and the filename (*results.log*). If *results.log* already exists in the target directory, you should *append* the new data rather than overwrite the existing file. If the file does not exist, your program should create it. If the directory specified by the *filepath* 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.
```

If the *filepath* variable is not set by the command-line flag, your program must print the results of algorithm audits by default to the *'results.log'* file in the default directory. The results must be saved in ASCII code, *i.e.*, human-readable.

4 Import Scenarios from a Configuration File (10 points)

Instead of generating scenarios solely randomly, you need to make sure in this task that your program can import scenarios from a data file. This will allow you to run audits on a consistent set of scenarios. In this task, you need to extend your *EthicalEngine* class to allow it to create scenarios based on data it reads from a configuration file.

4.1 Specify the Configuration File as Command-Line Argument

The config file should be specified when your program is launched. In this task, you need to create a command-line option. Command-line options or so-called *flags* specify options that modify the operation of your program. Options follow the program execution command on the command-line, separated by spaces. Options can be specified *in any order*. The following program calls are equivalent and should be supported by your program:

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

The command line argument following the flag $--config$ or $-c$ respectively specifies the filepath where the configuration file (*config.csv*) 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.
```

Default behaviour: If your program is run without specifying a config file (and without indicating the interactive mode), your program should run an algorithm audit with 100 truly, randomly generated scenarios writing its results to *results.log* in the default folder.

4.2 Parsing the Configuration File

Next, your program needs to read in the config file. Table 1 lists the contents of *config.csv*, a so-called **comma-separated values** (CSV) file. The file contains a list of values, each separated by a comma.
As can be seen in Table 1, 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 *Persona*. 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 Figure 1. The second scenario describes an illegal crossing with 4 pedestrians (2 animals) and 2 car passengers.
Your *EthicalEngine* class needs to be able to read in a config file as depicted in Table 1 and create a *Scenario* instance for each scenario the file contains. 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 format with the columns ordered as shown in Table 1.

4.3 Handle 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 number of data fields per row: in case the number of values in one row is less than or exceeds 10 values a **InvalidDataFormatException** should be thrown. Your program should handle such exceptions by issuing the warning statement "WARNING: invalid data format in config file in line $<$ *linecount* $>$" to the command-line and skip the respective row then continue reading in the next line.

2. Invalid data type: in case the value can not be casted 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 warning statement "WARNING: invalid number format in config file in line $<$ *linecount* $>$" to the command-line, assign a default value instead, and continue with the next value in that line.

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 a warning statement "WARNING: invalid characteristic in config file in line $<$ *linecount* $>$" to the command-line, assign a default value instead, and continue with the next value in that line.

Note that $<$ *linecount* $>$ depicts the line number in the config file where the error was found. While you can import the *NumberFormatException* from the package *java.lang* you will need to create custom exceptions for the other two.

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

Table 1: Contents of the configuration file (*config.csv*).

4.4 Audit your Algorithm Using the Scenarios from the Config File

Once your program has imported all scenarios from *config.csv* it should create a new *Audit*. Therefore, you need to extend your *Audit* class by adding two more methods:

- the constructor *Audit(Scenario[] scenarios)*: this constructor creates a new instance with a fixed set of scenarios.
- the public method *run()*: runs the simulation with the scenarios specified and runs each scenario through the *EthicalEngine* using its *decide(Scenario scenario)* method.

Use the *printToFile(String path)* method to show the results of your audit on the console-line as well as save your audit results to *'results.log'* or whatever is specified as a target *filepath* by the command-line flag $-r$ (or $--results$). The program should terminate after showing the statistic.

5 Interactive Scenarios (10 points)

Now it is time to let the user take over and be the judge. Therefore, you need to build an interactive console program, which presents the user with a number of ethical scenarios. These scenarios can either be randomly generated or imported from a config file. For each scenario the user is asked to make a decision about who should survive. The results are logged to a user file (*user.log*) but only if the user consents to it.

5.1 Program Setup

As described in Section 4.1, we will use command-line options or so-called *flags* to initialize the execution of *EthicalEngines*. Therefore, you should add a few more options as possible command-line 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 call should invoke the help:

```
$ java EthicalEngine --help
and
$ 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        Print Help (this message) and exit
   -r or --results     Optional: path to results log file
   -i or --interactive Optional: launches interactive mode
```

The help should be displayed when the $--help$ or $-h$ flag is set or if the $--config$ flag is set without an argument (*i.e.*, no path is provided). If the $--config$ or $-c$ flag is not set, your program should generate random scenarios. The flag $--interactive$ or $-i$ indicates the interactive user mode. Without this flag the audit from Section 3 should kick in. Only if the $--interactive$ or $-i$ is set the program launches its interactive scenarios. The flag $--results$ or $-r$ indicates the path and filename where the results of your audit should be saved to. See Section 3.2 for details.
The following command will launch the program with a config file in the interactive mode:

```
$ java EthicalEngine -i -c config.csv
```

Here is an example of launching the program in the interactive mode with random scenarios:

```
$ java EthicalEngine -i
```

5.2 Program Execution

You need to extend the *EthicalEngine* class to manage the user interaction and support the following program flow: show a welcome message, collect user consent for data collection, present 3 scenarios and have user judge these, show the statistic, and ask for another round of scenarios.

5.2.1 Show Welcome Screen

At the start of the program, a welcome message must be shown: your program should read in and display the contents of *welcome.ascii* to the user without modifying it. The message provides background information about Moral Machines and walks the user through the program flow.
Next, 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. After the welcome message, you program should therefore prompt the user with the following method 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 the user statistic to *'user.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:

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

5.2.2 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. Therefore, you should use the *Audit* class to keep track of the scenarios and decisions. Make sure to set the audit type to *User* using the method *setAuditType(String name)*. Scenarios are presented one by one using the *toString()* method of the *Scenario* instance and printing its outputs to the command-line. Each scenario should be followed by a prompt saying:

```
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 the 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 the 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.

5.2.3 Show the Statistic

The statistic must be printed to the command-line using the same method and format as described in Section 3.1. If the user previously consented to the data collection, the statistic is saved (*i.e.,* appended) to the file *user.log* using the function *printToFile(String filepath)* of the *Audit* class. Additionally, the user should be prompted to either continue or quit the program as follows:

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

Should the user choose *no* the program terminates after the following prompt:

```
That's all. Press Enter to quit.
```

As soon as the Enter key is pressed, the program should terminate. If the user decides to continue (*yes*), the next three scenarios should be shown. If the config file does not contain any more scenarios, the final statistic should be shown followed by the exit prompt (see above).
Here is an example of a statistic followed by a prompt to continue:

```
======================================
# User Audit
======================================
- % SAVED AFTER 3 RUNS
cat: 1.00
pregnant: 1.00
you: 1.00
senior: 0.67
student: 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 quit.
```

And that's it. Almost.

6 Documentation (5 points)

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. Only your UML diagram needs to be submitted through Canvas. You do not need to submit your documentation created through JavaDoc. This will be created automatically.

6.1 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 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.

Make sure to save your UML diagram in PDF format and add it to your Github repository's root directory as *UML.pdf*.

6.2 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 tag

You can leave minor helper function that you may have added. Make sure to test the correct generation of your documentation using javadoc[3].

[3] https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html

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

7 Reflection (Bonus Task)

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 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** you may have lost somewhere else. Note that you cannot exceed the total of 60 points for the entire project. But we would love to read about your thoughts.

8 Testing Before Submission

You will find two types of local tests available: 1) an invocation test with some basic assertions and 2) a program input/output test. The tests along with the sample input, configuration files, and the expected output can be found in the template repository along with more detailed instructions on how to run them.

8.1 Test Runner

In the repository, you will find a Java file called *TestRunner.java*. It comprises a range of methods to instantiate and invoke the basic functionality of your code. You can run it as follows:

```
javac TestRunner.java && java TestRunnner
```

While the main functionality of your code is still missing, these tests will lead to a range of compile errors. We recommend you to first finish your project code and then start debugging using these tests. Once you have addressed all compile errors (**don't change the tests, change your code**), the test runner will launch a series of comparisons between your program's and the expected output. These tests will be the basis for the automated grading of your project, so make sure to pass them as well as you can. Passing them, however, does not guarantee the completeness of your program. You will need to thoroughly test your program beyond what these tests provide.

8.2 Scenario Import and Interactive Mode Test

To further test your program you will find a range of test inputs for your program in the repository.
To run the interactive mode test, follow these steps:

1. Check your Java code files and make sure you have the test input data files ready in your *tests* folder. `in_interactive_config_3` provides user input (interactive mode) if you run your program with the `config_3.csv` file.

2. Open a console, navigate to your project directory (where your .java classes reside), and compile your program: `javac *.java`. This command will compile all your java file in the current folder.

3. Run your program with the following command-line parameters:

```
java EthicalEngine -i -c tests/config_3.csv < tests/in_interactive_config_3 > output
```

This command will run the EthicalEngine main method in interactive mode with user input described in '`in_interactive_config_3`" as input and write the output to a file called `output`)

4. Inspect the file `output` as it may contain any errors your program execution may have encountered.

5. Compare your result with the provided output file `out_interactive_config_3`. Fix your code if they are different.

It is crucial that **your output follows exactly the same format shown in the examples in this document.**

NOTE: The test cases used to mark your submissions will be different from the sample tests given. You should test your program extensively to **ensure it is correct for other input values** with the same format as the sample tests.

9 Submission

Your submission must have all required files as described in Section 1 plus the UML file and any additional files you may have created that are necessary to run your program. If your program does not compile under Java 1.8 you run the risk of getting significantly marked down. If you correctly cloned the assignment repository, some mandatory skeleton files should already be in your working directory.

Make sure to submit all files, including those your package directory (*ethicalengine*) and leave the *welcome.ascii* message in your root project folder as depicted in Section 1. The entry point of your program should be in the class called *EthicalEngine* (in the file called EthicalEngine.java). Thus, your program will be invoked via:

```
java EthicalEngine
```

Make sure the command-line options work as described in Section 5.1. **Note the exact spelling including lower and upper case.** The same goes for the classes in the package *ethicalengine* depicted in Section 1. You should verify your submission locally as described above before submitting your code to GitHub.

Make sure to fill in your information into the *authorship.txt* declaration, which is included in the skeleton code. If you fail to do so, there is a chance we cannot allocate your submission and you run the risk of scoring 0 on the final project. Also, make sure to include your authorship in all core classes as comments.

You can edit and re-submit your code to GitHub as many times you want as long as you submit before the submission deadline of the assignment.

A good practice to test your code is to clone your repository into a fresh directory and try and run your program there.

The keyword `import` is available for you to use standard java packages. However, make sure that your project structure follows the structure as described in Section 1. **DO NOT** use any other packages than *ethicalengine*. Note that the files *Audit.java* and *EthicalEngine.java* **ARE NOT** part of the *ethicalengine* package and must therefore be placed outside of the *ethicalengine* directory (into your default directory). Adding any other packages to your project will break the automated tests and you will lose points. If you are using Netbeans as the IDE, be aware that the project name may automatically be used as the package name. You must remove these package names at the beginning of the source files before you submit them to the system.

Make sure to use **ONLY ONE** Scanner object throughout your program. Otherwise the automatic test will cause your program to generate exceptions and terminate. The reason is that in the automatic test, multiple lines of test inputs are sent all together to the program. As the program receives the inputs, it will pass them all to the currently active Scanner object, leaving the rest Scanner objects nothing to read and hence cause run-time exception. Therefore it is crucial that **your program has only one Scanner**

object. Arguments such as "It runs correctly when I do manual test, but fails under automatic test" will not be accepted.

The deadline for the project is **5pm (AEDT), Friday 20 Nov, 2020**.

What will be graded? The **last** version of your program committed before the submission deadline.

9.1 Late-Submission Penalties

There is a 20% penalty per day for late submissions.

For example, suppose your project gets a mark of 50 but changes are submitted within 1 day after the deadline, then you receive a **20% penalty** and the mark will be 40 after the penalty.

Any updates to your code on GitHub made after the submission deadline will incur a late-submission penalty. This means that even if you make a minor edit and re-submit a file after the deadline your entire submission will be subject to a late-submission penalty! Be aware of timezone differences! Submissions via email will not be accepted.

There will be **0 marks for your submission if you make changes after the 5pm (AEDT), Friday 23 Nov, 2020**.

10 Certification of Individual Work

Note well that this project is your final assessment, so cheating is out of the question. Any form of material exchange, whether written, electronic or 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 exchange of more than basic ideas, formal disciplinary action will be taken for all involved parties. A sophisticated program that undertakes deep structural analysis of Java code identifying regions of similarity will be run over all submissions in "compare every pair" mode.

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

Cheating is not worth it. Trust me, it never is.

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!