OGP Assignment 2018-2019: **Jumping Alien** (Part I)

This text describes the first part of the assignment for the course *Object-oriented Programming*. There is no exam for this course, so all grades are scored on the assignment. The assignment is preferably made in groups consisting of two students. It is, however, possible to do the project individually. Each team must send an email containing the names and the course of studies of all team members to ogp-project@cs.kuleuven.be before March 1, 2019. If you cooperate, only one member of the team should send an email putting the other member in CC.

If during the semester conflicts arise within a group, this should be reported to ogp-project@cs.kuleuven.be and each of the group members is then required to complete the project on their own.

The assignment consists of three parts. The first part focusses on a single class, the second part on associations between classes, and the third part on inheritance and generics. After handing in the third part, the entire solution must be defended before Professor Steegmans or Professor Jacobs.

A number of teaching assistants (TAs) will advise the students and answer their questions. More specifically, each team has a number of hours where the members can ask questions to a TA. The TA plays the role of consultant who can be hired for a limited time. In particular, students may ask the TA to clarify the assignment or the course material, and discuss alternative designs and solutions. However, the TA will not work on the assignment him/herself. Consultations will generally be held in English. Thus, your project documentation, specifications, and identifiers in the source code should be written in English. Teams may arrange consultation sessions by email to ogp-project@cs.kuleuven.be. Please outline your questions and propose a few possible time slots when signing up for a consultation appointment.

To keep track of your development process, and mainly for your own convenience, we encourage you to use the *Git* version control system. Instructions on how to obtain a private repository on GitHub, already populated with the provided GUI code (see section 3), as well as a short tutorial, will appear in a separate document on Toledo.

The goal of this assignment is to test your understanding of the concepts introduced in this course. For that reason, we provide a graphical user interface and it is up to the teams to implement the requested functionality. The requested functionality is described at a high level in this document and it is up to the student to design and implement one or more classes that provide this functionality. The grades for this assignment do not depend only on functional requirements. We will also pay attention to documentation, accurate specifications, re-usability and adaptability.

1 Assignment

This assignment aims to create a platform video game that is loosely based on the Super Mario series by Nintendo. In **Jumping Alien**, the player controls a little green character called `Mazub`. The goal of the game is to move `Mazub` safely through a hostile two-dimensional game world, avoiding or destroying enemies and collecting items. In this first part of the assignment we focus on a single class `Mazub` that implements the player character with the ability to jump and to run to the left and to the right in a simplistic game world. Of course, your solution may contain additional helper classes (in particular classes marked *@Value*). In the remainder of this section, we describe the class `Mazub` in more detail. Importantly, all aspects of your implementation of the class `Mazub` shall be specified both formally and informally. In the second and third part of the assignment, we will extend `Mazub` and add additional classes to our game.

1.1 The Game World

Jumping Alien is played in a rectangular game world that is composed of a fixed number of $X = 1024$ times $Y = 768$ adjointly positioned, non-overlapping *pixels*. Each pixel is located at a fixed position (x, y). The position of the bottom-left pixel of the game world shall be $(0, 0)$. The position of the top-right pixel of the game world shall be $(x_{max}, y_{max}) = (X - 1, Y - 1)$. All pixels are square in shape. For the purpose of calculating locations, distances and velocities of game objects, each pixel shall be assumed to have a side length of $1\,\mathrm{cm} = 0.01\,\mathrm{m}$.

1.2 The Class `Mazub`

The player character `Mazub` is a rectangular object of varying size that can move within a game world.

1.2.1 Position

As illustrated in Figure 1, `Mazub` occupies an area of X_P times Y_P pixels of a game world and `Mazub`'s position is given as (x, y), denoting the pixel of the game world that is occupied by `Mazub`'s bottom-left pixel. `Mazubs` should never be positioned completely outside the game world. More precisely, the position of the bottom-left pixel must always be inside the boundaries of the world. Notice that this does not prevent `Mazubs` to partially cross the right and top border of the game world.

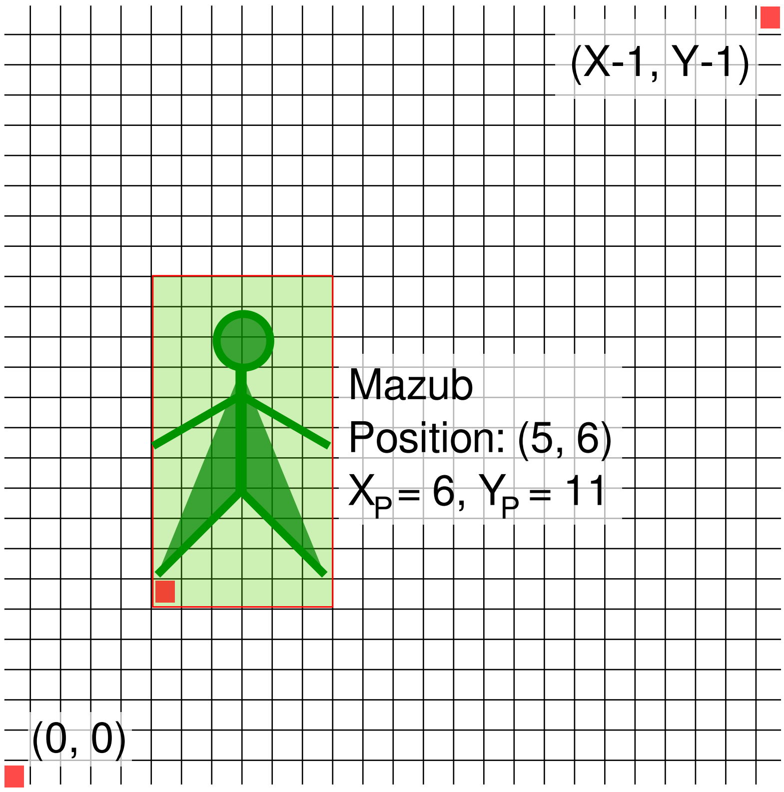

Figure 1: **Jumping Alien**: The game world and the player character `Mazub`.

When displaying `Mazubs`, their position will always be aligned with pixels of their game world. In other words, `Mazubs` will not be displayed as partially occupying some pixels of the game world. They either fully occupy a game world's pixel, or they do not occupy such a pixel at all. Nevertheless, the actual position of a `Mazub` is the position of its bottom-left corner in the game world. That actual position is given as (x, y), in which both x and y are double precision floating-point values expressing the distance in meters of the bottom-left corner of `Mazub` to the bottom-left corner of the game world. The actual position of `Mazub` is not aligned with the game world, meaning that the actual position can be any position within the pixel in the game world occupied by the left-bottom pixel of `Mazub`. For example, if the actual position of Mazub in the world is $(1.234\,\mathrm{m}, 5.678\,\mathrm{m})$, its corresponding pixel position is $(123, 567)$.

All aspects concerning the position of `Mazub` shall be worked out defensively.

1.2.2 Orientation

`Mazub` is oriented either to the left, to the right or to the front. When `Mazub` is moving, its orientation will be either to the left or to the right. When `Mazub` is not moving, and at least 1 second has passed since its last move has ended, `Mazub` will be oriented to the front.

All aspects concerning the orientation of `Mazub` shall be worked out nominally.

1.2.3 Horizontal velocity

`Mazub` is able to move horizontally in both directions. The horizontal velocity is negative if `Mazub` is moving to the left; it is positive if `Mazub` is moving to the right. The horizontal velocity of `Mazub` is expressed in meters per seconds and is constrained by a minimum value and a maximum value. If `Mazub` is standing straight while moving horizontally, the magnitude of its horizontal velocity (= absolute value) will not be below a minimum value of $1.0\,\mathrm{m/s}$ and will not exceed a maximum value of $3.0\,\mathrm{m/s}$. If `Mazub` is ducking while moving horizontally, its horizontal velocity will have a constant value that is equal to $1.0\,\mathrm{m/s}$ or $-1.0\,\mathrm{m/s}$ depending on the direction in which `Mazub` is moving. Obviously, if `Mazub` is not moving horizontally, its horizontal velocity is equal to $0.0\,\mathrm{m/s}$.

The minimum value and the maximum value for the horizontal velocity do not change during the game. However, in the future, the actual values for each of them may change both in case `Mazub` is standing straight and in case `Mazub` is ducking. The minimum value for the horizontal velocity will never be changed to a value below $1.0\,\mathrm{m/s}$. The maximum value for the horizontal velocity will never be changed to a value below the minimum horizontal velocity. In future versions of the game, it must be possible to have different `Mazub`s with different minimum values and maximum values for their horizontal velocity.

All aspects concerning the horizontal velocity of `Mazub` shall be worked out totally.

1.2.4 Vertical velocity

Mazub is able to move vertically in both directions. The vertical velocity of Mazub is expressed in meters per seconds and will never exceed $8\,\mathrm{m/s}$. That value will always be the same for all Mazubs, and will never change in future versions of the game. A positive value for the vertical velocity means that Mazub is moving upwards. With a negative vertical velocity, Mazub falls down. There is no specific lower bound for the vertical velocity. Obviously, if Mazub is not moving vertically, its vertical velocity is equal to $0.0\,\mathrm{m/s}$.

All aspects concerning the vertical velocity of Mazub shall be worked out totally.

1.2.5 Running

The player character may run to the right side or to the left side of the game world. The class Mazub shall provide methods `startMove` and `endMove` to initiate, respectively to stop movement in a given direction. Obviously, you can not start moving if you are already moving, and you can not stop moving if you are not already moving. These methods must be worked out nominally.

Once `startMove` has been invoked, Mazub starts moving in the given direction with a horizontal velocity equal to its minimum value, accelerating with $a_x = 0.9\,\mathrm{m/s^2}$ in that direction until the velocity reaches its maximum value. Mazub's horizontal velocity after some Δt seconds can be computed as $v_{x_{new}} = v_{x_{current}} + a_x \Delta t$, where $v_{x_{current}}$ is Mazub's current horizontal velocity. Once v_x equals its maximum value, Mazub's horizontal velocity shall remain constant at that maximum value. The velocity of Mazub shall drop to zero immediately as `endMove` is invoked.

It must be possible to change the horizontal acceleration in future versions of the game. All Mazubs will always have the same value for the horizontal acceleration.

1.2.6 Jumping and Falling

The player character may also *jump*. Similar to running, the class Mazub shall provide methods `startJump` and `endJump` to initiate or stop jumping. These methods must be worked out defensively. Obviously, Mazub can not start jumping if it is already jumping, and Mazub can not stop jumping if it is not already jumping.

Once `startJump` has been invoked, Mazub starts moving with a velocity of $v_y = 8\,\mathrm{m/s}$ in positive y-direction (i.e. upwards). Invoking the method `endJump` shall set Mazub's vertical velocity to zero if the current vertical velocity is greater than zero.

Whenever `Mazub`'s y-position is not zero, i.e., `Mazub` is not located at the bottom of the game world, `Mazub` shall *fall*. More specifically, `Mazub` shall accelerate with $a_y = -10\,\mathrm{m/s^2}$ in positive y-direction until `Mazub`'s y-position reaches zero.

The values for the initial vertical velocity and the vertical acceleration do not change during the game, and will not change in future versions of the game.

1.2.7 Ducking

The player character may also duck so as to decrease its size. In future phases of the game this will be used to avoid enemies or to access narrow passages. `Mazub` can duck while standing still, while moving or while jumping. The class `Mazub` shall provide methods `startDuck` and `endDuck` to initiate and stop ducking. These methods are are to be implemented totally. Ducking affects the X_P and Y_P attributes of `Mazub` as explained in Section 1.2.9. Ducking also restricts v_{xmax} to $1\,\mathrm{m/s}$. In other words, `Mazub` moves at a constant horizontal velocity as long as `Mazub` is ducking.

1.2.8 AdvanceTime

The class `Mazub` shall provide a method `advanceTime` to update the position and the velocity of `Mazub` based on its current position, its current velocity, its current acceleration and a given time duration Δt in seconds. This duration Δt shall never be less than zero and always be smaller that $0.2\,\mathrm{s}$. The method `advanceTime` will also update the image to be used for displaying `Mazub`, as we will explain in section Section 1.2.9.

The horizontal velocity of `Mazub` may be computed as $v_{x_{new}} = v_{x_{current}} + a_x \Delta t$, in which $v_{x_{current}}$ denotes the current horizontal velocity of `Mazub`, and a_x its horizontal acceleration. The horizontal position of `Mazub` may be computed as $x_{new} = x_{current} + v_{x_{current}} \Delta t + \frac{1}{2} a_x \Delta t^2$, in which $x_{current}$ denotes the current horizontal position of `Mazub`. Similar formulae hold for computing the vertical velocity and the vertical position of `Mazub` after some period of time. The method `advanceTime` must ensure that the bottom-left pixel of `Mazub` stays at all times within the boundaries of the game world.

The method `advanceTime` shall be worked out using total programming. You may safely assume that no in-game time elapses between the last invocation of `advanceTime` and a subsequent invocation of the methods `startMove`, `endMove` or any of the `start...` and `end...` methods specified in this section.

Table 1: Association between sprite index and character behaviour.

Index	To be displayed if `Mazub`...
0	is not moving horizontally, has not moved horizontally within the last second of in-game time and is not ducking.
1	is not moving horizontally, has not moved horizontally within the last second of in-game time and is ducking.
2	is not moving horizontally but its last horizontal movement was to the right (within 1s), and the character is not ducking.
3	is not moving horizontally but its last horizontal movement was to the left (within 1s), and the character is not ducking.
4	is moving to the right and jumping and not ducking.
5	is moving to the left and jumping and not ducking.
6	is ducking and moving to the right or was moving to the right (within 1s).
7	is ducking and moving to the left or was moving to the left (within 1s).
$8..(8+m)$	the character is neither ducking nor jumping and moving to the right.
$(9+m)..(9+2m)$	the character is neither ducking nor jumping and moving to the left.

1.2.9 Character Size and Animation

To display the player character in a visualisation of the game world, `Mazub` will have an array of sprites. A sprite is an image of X_P times Y_P pixels. A class `Sprite` is provided with the assignment, which offers the methods `getHeight` and `getWidth` to inspect the size of a sprite. The constructor of `Mazub` shall have a parameter to accept an array of `sprites`. That array must contain an even number of at least 10 images. Moreover, the array of sprites must have an equal number of images for running to the left and for running to the right.

The class `Mazub` shall further provide a method `getCurrentSprite` that returns the image to be used for displaying the player character. The image to be displayed must reflect the current state of the player character, as described in Table 1. If there are multiple such images (i.e., $m > 0$), these images shall be used alternatingly. Starting with the image $\texttt{images}_i$ with the smallest i appropriate for the current action, a different image shall be selected every 75 ms. As `Mazub` continues to run, $\texttt{images}_{i+1}$ shall be displayed, followed by $\texttt{images}_{i+2}$, and so on. Once the set of images for the current action is exhausted, i.e., $\texttt{images}_{i+m}$ has been displayed and `Mazub` is still running, the above procedure repeats starting from $\texttt{images}_i$.

Importantly, each $\texttt{images}_i$ may have different dimensions. Independently of whether `getCurrentSprite` is invoked, `Mazub`'s X_P and Y_P must always be reported by the inspectors as the dimensions of the image that is appropriate with respect to `Mazub`'s current state. For your implementation and for the documentation you may safely assume that no in-game time elapses between the last invocation of `advanceTime` and a subsequent invocation of the methods `getCurrentSprite` or `getHeight` and `getWidth`.

All aspects concerning sprites must be worked out in a defensive way. No documentation is required for methods or aspects of methods related to sprites.

2 Storing and Manipulating Real Numbers as Floating-Point Numbers

In your program, you shall use type **double** as the type for variables that conceptually need to be able to store arbitrary real numbers, and as the return type for methods that conceptually need to be able to return arbitrary real numbers.

Note, however, that variables of type **double** can only store values that are in a particular subset of the real numbers (specifically: the values that can be written as $m \cdot 2^e$ where $m, e \in \mathbb{Z}$ and $|m| < 2^{53}$ and $-1074 \leq e \leq 970$), as well as positive infinity (written as `Double.POSITIVE_INFINITY`) and negative infinity (written as `Double.NEGATIVE_INFINITY`). (These variables can additionally store some special values called *Not-a-Number* values, which are used as the result of operations whose value is mathematically undefined such as 0/0; see method `Double.isNaN`.) Therefore, arithmetic operations on expressions of type **double**, whose result type is also **double**, must generally perform *rounding* of their mathematically correct value to obtain a result value of type **double**. For example, the result of the Java expression `1.0/5.0` is not the number 0.2, but the number[1]

$$0.200000000000000011102230246251565404236316680908203125$$

When performing complex computations in type **double**, rounding errors can

[1] You can check this by running `System.out.println(new BigDecimal(1.0/5.0))`.

accumulate and become arbitrarily large. The art and science of analysing computations in floating-point types (such as **double**) to determine bounds on the resulting error is studied in the scientific field of *numerical analysis*.

However, numerical analysis is outside the scope of this course; therefore, for this assignment we will be targeting not Java but *idealised Java*, a programming language that is entirely identical to Java except that in idealised Java, the values of type **double** are exactly the extended real numbers plus some nonempty set of *Not-a-Number* values:

$$\mathbf{double} = \mathbb{R} \cup \{-\infty, +\infty\} \cup \mathit{NaNs}$$

Therefore, in idealised Java, operations in type **double** perform no rounding and have the same meaning as in regular mathematics. Your solution should be correct when interpreting both your code and your formal documentation as statements and expressions of idealised Java.

So, this means that for reasoning about the correctness of your program you can ignore rounding issues. However, when testing your program, of course you cannot ignore these. The presence of rounding means that it is unrealistic to expect that when you call your methods in your test cases, they will produce the exact correct result. Instead of testing for exactly correct results, it makes more sense to test that the results are within an acceptable distance from the correct result. What "acceptable distance" means, depends on the particular case. For example, in many cases, for a nonzero expected value, if the relative error (the value $|r-e|/|e|$ where r and e are the observed and expected results, respectively) is less than 0.01%, then that is an acceptable result. You can use JUnit's `assertEquals(double, double, double)` method to test for an acceptable distance.

3 User Interface

We provide a graphical user interface (GUI) to visualise the effects of various operations on `Mazub`. The user interface is part of the Eclipse project that you can get from GitHub. You will find a folder `src-provided` that contains the source code of the user interface, the `Util` and `Sprite` class and further helper classes. Generally, the files in this folder require no modification from your side. The classes that you develop must be placed in the folders `src` (implementation classes) and `tests` (test classes).

To connect your implementation to the GUI, write a class `Facade` in package `jumpingalien.facade` that implements `IFacade`. `IFacade.java` contains additional instructions on how to implement the required methods. Read this documentation carefully.

To start the program, you may execute the `main` method in the provided class `jumpingalien.JumpingAlien`. After starting the program, you can press keys to modify the state of the program. Commands are issued by pressing the `left`, `right`, `up` and `down` arrow keys to start running to the left, right, and to start jumping and ducking, respectively. That is, pressing the above keys will invoke `startMoveLeft`, `startMoveRight`, `startJump` or `startDuck` on your Facade. Releasing these keys invokes `endMoveLeft`, `endMoveRight`, `endJump` and `endDuck` accordingly. Pressing `Esc` terminates the program.

You can freely modify the GUI as you see fit. However, the main focus of this assignment is the class `Mazub`. No additional grades will be awarded for changing the GUI.

4 Testing

We will test that your implementation works properly by running a number of JUnit tests against your implementation of `Facade`. As described in the documentation of `IFacade`, the methods of your `Facade` class shall only throw `ModelException`. The test suite that we will use to evaluate part 1 is distributed along with the assignment. If you run the suite, it will yield a score that will be integrated in your final score for the course.

5 Submitting

The solution must be submitted via Toledo as a jar file individually by all team members before the 10th of March 2019 at 11:59 PM. You can generate a jar file on the command line or using eclipse (via export). Include all source files (including tests) and the generated class files. Include your name, your course of studies and a link to your code repository in the comments of your solution. When submitting via Toledo, make sure to press OK until your solution is submitted!

6 Feedback

A TA will give feedback on the first part of your project. These feedback sessions will take place between the 18th and the 29st of March. More information will be provided via Toledo.