OGP Assignment 2014-2015: **Jumping Alien** (Part I)

This text describes the first part of the assignment for the course *Object-oriented Programming* (OGP). There is no exam for this course. Therefore, all grades are scored on this assignment. The assignment is preferably taken in groups consisting of two students; only in exceptional situations the assignment can be worked on individually. Each team must send an email containing the names and the course of studies of all team members to ogp-inschrijven@cs.kuleuven.be before the 1st of March 2015. 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-inschrijven@cs.kuleuven.be and each of the group members is then required to complete the project on their own.

In the course of the assignment, we will create a simple game that is loosely based on the *Super Mario* series of platform video games by Nintendo. Note that several aspects of the assignment will not correspond to any of the original games by Nintendo. In total, the assignment consists of three parts. The first part focusses on a single class, the second on associations between classes, and the third on inheritance and generics.

The goal of this assignment is to test your understanding of the concepts introduced in the course. For that reason, we provide a graphical user interface for the game and it is up to the teams to implement the requested functionality. This functionality is described at a high level in this document and the student may design and implement one or more classes that provide this functionality, according to their best judgement. Your solution should be implemented in Java 8, satisfy all functional requirements and follow the rules described in this document. The assignment may not answer all possible questions you may have concerning the system itself (functional requirements) or concerning the way it should be worked out (non-functional requirements). You are free to fill in those details in the way that best suits your project. As an example, if the assignment does not impose to use nom-

inal programming, total programming or defensive programming in working out some aspect of the game, you are free to choose the paradigm you prefer for that part. The ultimate goal of the project is to convince us that you master all the concepts underlying object-oriented programming. The goal is not to hand it the best possible Super Mario-like game. Therefore, the grades for this assignment do not depend only on correctly implementing functional requirements. We will pay attention to documentation, accurate specifications, re-usability and adaptability. After handing in your solution to the third part of this assignment, the entire solution must be defended in front of Professor Steegmans.

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 itself. 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 a source code management and revision control system such as *Subversion* or *Git*.

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 trough 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 run to the left and 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. For your support, you will be provided a JAR file containing the user interface for the game together with some helper classes.

1.1 The Game World

Jumping Alien is played 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 $cm = 0.01$ m.

1.2 The Class `Mazub`

The player character `Mazub` is a rectangular object that can move within a game world. As illustrated in Figure 1, `Mazub` occupies a 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. `Mazub`s 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 `Mazub`s to partially cross the right and top border of the game world. In the remainder of this section we explain how and when `Mazub`'s position as well as its size (cf. Sec. 1.2.4) changes during the execution of the game. As the size and position of `Mazub` refer to the positions of pixels in the game world, these aspects of the class `Mazub` shall be worked out defensively using integer numbers. The class `Mazub` shall provide methods to inspect the player character's position and dimensions.

1.2.1 Running

The player character may run to the right or left side of the game world. The class `Mazub` shall provide methods `startMove` and `endMove` to initiate or stop movement in a given direction. These methods must be worked out nominally. The direction is restricted to positive and negative x-direction (i.e. right and left). Once `startMove` has been invoked, `Mazub` starts moving with a horizontal velocity of $v_x = 1m/s$ in the given direction, accelerating with $a_x = 0.9m/s^2$ in that direction, up to a maximum velocity of $v_{x_{max}} = 3m/s$. `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 $v_{x_{max}}$, `Mazub`'s horizontal velocity shall remain constant at $v_{x_{max}}$. The velocity of `Mazub` shall drop to zero immediately as `endMove` is invoked.

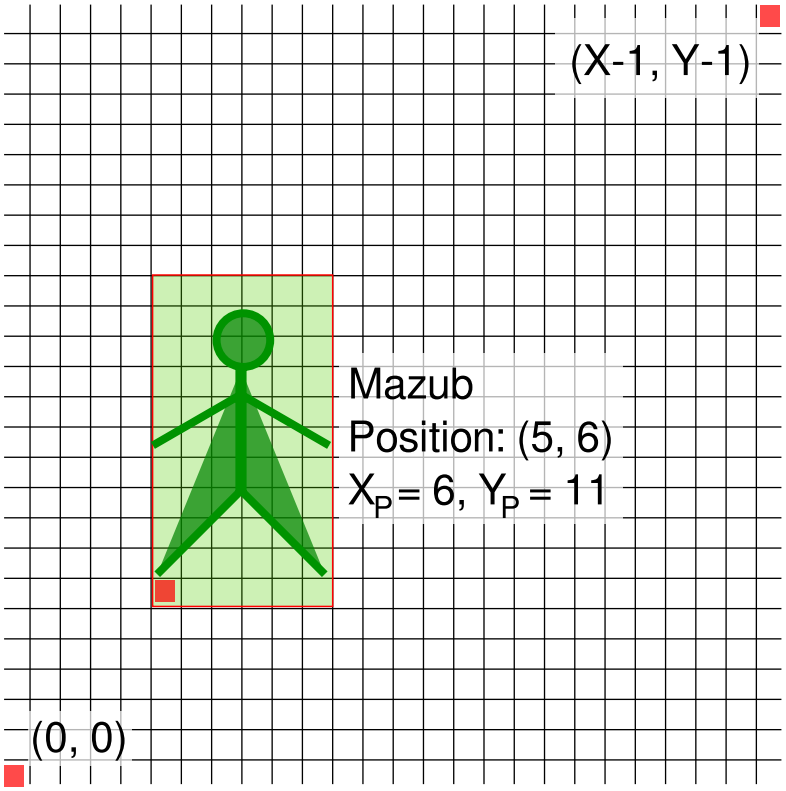

Figure 1: **Jumping Alien**: The game world and the player character `Mazub`.

The values for the initial horizontal velocity, the maximum horizontal velocity and the horizontal acceleration do not change during the game. However, in the future, the actual values for each of them may change. The initial velocity will never be changed to a value below 1m/s. The maximum velocity will never be changed to a value below the initial velocity. However, it must then be possible to have different `Mazub`s with different values for the initial and the maximum horizontal velocities. The acceleration may change in both directions, without ever becoming negative. All `Mazub`s will always have the same value for the horizontal acceleration. Finally, there is no requirement to support other units (cm, m/s, ...) than the ones used in this text.

The class `Mazub` shall provide a method `advanceTime` to update the position and velocity of `Mazub` based on the current position, velocity, acceleration and a given time duration Δt in seconds. This duration Δt shall never be less than zero and always be smaller that $0.2s$. The distance travelled by `Mazub` may be computed as $s = v\Delta t$ while the velocity is constant, or as $s = v_{x_{current}}\Delta t + \frac{1}{2}a_x\Delta t^2$ if acceleration is involved. The x-component of `Mazub`'s new position is then computed as $x_{new} = x_{current} + s$. It must

be possible to turn to other formulae to compute the horizontal distance travelled by a `Mazub` during some period of time. Those formulae will never yield a value that exceeds the value described by the formula above. The method `advanceTime` shall be worked out using defensive programming. All methods that concern the horizontal acceleration of `Mazub` shall be worked out using total programming. 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 `startMove`, `endMove` or any of the `start...` and `end...` methods specified in this section. The method `advanceTime` must ensure that the bottom-left pixel of `Mazub` stays at all times within the boundaries of the game world.

All characteristics of the class `Mazub` that concern the player object's velocity, acceleration and timing must be treated as double precision floating-point numbers. That is, use Java's primitive type `double` to compute and store these attributes. The characteristics of `Mazub` must at all times be valid numbers (meaning that `Double.isNaN` returns `false`). However, we do not explicitly exclude the special values `Double.NEGATIVE_INFINITY` and `Double.POSITIVE_INFINITY` (unless specified otherwise). All characteristics of the class `Mazub` that concern the player object's position in the game world must be worked out using integer numbers. Positions may be computed as double precision floating-point numbers but shall be rounded down (i.e. $x = \lfloor x_{new} \rfloor$) to a multiple of the size of a game world pixel to determine `Mazub`'s effective position in the game world. Note that for small Δt and low velocities, `Mazub`'s effective position in the game world may not change although `Mazub`'s actual position with respect to game physics (as described in this section) does change and must be considered in future computations.

In addition to the methods specified above, class `Mazub` shall provide methods to inspect the player character's orientation (i.e. left or right), horizontal velocity and horizontal acceleration.

1.2.2 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, which must be worked out defensively. Once `startJump` has been invoked, `Mazub` starts moving with a velocity of $v_y = 8m/s$ 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.

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

reaches zero. All methods that concern the vertical acceleration of `Mazub` shall be worked out using total programming.

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.

Running and jumping may interleave and thereby influence each other. Thus, the specification of the jumping behaviour extends the `advanceTime` method with vertical movement. Similar to horizontal movement, this vertical movement generally follows the game physics described in Section 1.2.1: the vertical component of `Mazub`'s current velocity may be computed as $v_{y_{new}} = v_{y_{current}} + a_y \Delta t$. Likewise the y-component of `Mazub`'s position may be computed as $y_{new} = y_{current} + v_{y_{current}} \Delta t + \frac{1}{2} a_y \Delta t^2$. Again, it must be possible to turn to other formulae to compute the vertical distance travelled by a `Mazub` during some period of time. Those formulae will never yield a value that exceeds the value described by the formula above.

Again, characteristics of the class `Mazub` that concern the player object's velocity, acceleration and timing must be treated as double precision floating-point numbers. The class `Mazub` shall provide methods to inspect the player character's vertical velocity and vertical acceleration.

1.2.3 Ducking

The player character may also duck so as to decrease their size. In future phases of the game this will be used to avoid enemies or to access narrow passages. The class `Mazub` shall provide methods `startDuck` and `endDuck` to initiate and stop ducking, which are to be implemented defensively. Ducking affects the X_P and Y_P attributes of `Mazub` as explained in Section 1.2.4. Ducking also restricts v_{xmax} to $1 m/s$.

1.2.4 Character Size and Animation

To display the player character in a visualisation of the game world, the class `Mazub` shall provide a method `getCurrentSprite`, which is to be implemented using nominal programming. Importantly, formal documentation of the method `getCurrentSprite` is *not* required. `getCurrentSprite` shall return a `Sprite` of X_P times Y_P pixels that represents an image of the player character. A class `Sprite` is provided with the assignment, which offers the methods `getHeight` and `getWidth` to inspect the size of a sprite. The image to be displayed varies depending on actions currently performed by the player character. More specifically, the constructor of `Mazub` shall accept an array of `images` as a parameter. This array contains an even number of $n \geq 10$

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.

images such that $\texttt{images}_0$ refers to the first element and $\texttt{images}_{n-1}$ refers to the last element of the array. Depending on the current state of the player character, `getCurrentSprite` shall return a specific image $\texttt{images}_i$; a list of indices i together with a description of the character state is given in Table 1.

There will be an equal number of images for running to the left and to the right ($\texttt{images}_{8..(n-1)}$). If there are multiple such images (i.e., $m > 0$), these images shall be used alternating. Starting with the image $\texttt{images}_i$ with the smallest i appropriate for the current action, a different image shall be selected every $75ms$. 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 $\texttt{images}_i$. It must be possible to turn to other algorithms for displaying successive images of a `Mazub` during some period of time. As for the algorithm above, alternative algorithms will only use images that apply to the current range. The method

`advanceTime` must only implement the current algorithm. In this part of the project, the documentation must not specify the new image that applies to the `Mazub` at stake.

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`.

2 Reasoning about Floating-point Numbers

Floating-point computations are not exact. This means that the result of such a computation can differ from the one you would mathematically expect. For example, consider the following code snippet:

```
double x = 0.1;
double result = x + x + x;
System.out.println(result == 0.3);
```

The last statement outputs `false`, even though $0.1 + 0.1 + 0.1$ is mathematically equal to 0.3. The output is `false` because the variable `result` holds the value 0.30000000000000004.

A Java `double` consists of 64 bits. Clearly, it is impossible to represent all possible real numbers using only a finite amount of memory. For example, $\sqrt{2}$ cannot be represented exactly and Java represents this number by an approximation. Because numbers cannot be represented exactly, floating point algorithms make rounding errors. Because of these rounding errors, the expected outcome of an algorithm can differ from the actual outcome.

For the reasons described above, it is generally bad practice to compare the outcome of a floating-point algorithm with the value that is mathematically expected. Instead, one should test whether the actual outcome differs at most ϵ from the expected outcome, for some small value of ϵ. The class `Util` (included in the assignment) provides methods for comparing doubles up to a fixed ϵ.

The course *Numerieke Wiskunde* discusses the issues regarding floating-point algorithms in more detail. For more information on floating point numbers, we suggest that you follow the tutorial at
http://introcs.cs.princeton.edu/java/91float/.

3 Testing

Write JUnit test suite for the class `Mazub` that tests each public method. Include this test suite in your submission.

4 User Interface

We provide a graphical user interface (GUI) to visualise the effects of various operations on `Mazub`. The user interface is included in the assignment as a JAR file. When importing this JAR file 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.part1.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 class `JumpingAlienPart1`. 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.

We will test that your implementation works properly by running a number of JUnit tests against your implementation of `IFacade`. As described in the documentation of `IFacade`, the methods of your `IFacade` implementation shall only throw `ModelException`. An incomplete test class is included in the assignment to show you what our test cases look like.

5 Submitting

The solution must be submitted via Toledo as a JAR file individually by all team members before the 15th of March 2015 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` to confirm the submission!

6 Feedback

A TA will give feedback on the first part of your project. These feedback sessions will take place between the 23rd of March and the 3rd of April 2015 More information will be provided via Toledo.