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

This text describes the second 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. In principle, you should compile your solutions for the second and third parts of the assignment with the partner you chose for the first part. You are, however, allowed to start working with a new partner, or to work out the rest of the project on your own. Changes must be reported to ogp-inschrijven@cs.kuleuven.be before the 20th of March. 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 focussed on a single class, this second part emphasises on associations between classes, and the third part involves 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 nominal 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 the first part of the assignment we focused 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. In this second part we extend `Mazub`, introduce a more complex game world and further game objects that interact with `Mazub`. Thus, the focus of this part of the assignment is on associations between classes. To draw your attention we have highlighted new classes and alterations in the specification of `Mazub` in blue. Of course, your solution may contain additional helper classes (in particular classes marked *@Value*) and may already make use of concepts such as inheritance. In the remainder of this section, we describe the classes `World`, `Mazub`, `Plant`, `Slime` and `Shark` in more detail. All aspects of the class `Mazub` shall be specified both formally and informally. All aspects of the class `World` shall only be specified in a formal way. 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 Class `World`

Jumping Alien is played in a rectangular game world that is composed of X times Y 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 $1\ cm = 0.01\ m$.

Pixels are grouped together to `tiles`. All tiles are square and of the same size, given as the *length* of all the tiles' sides in pixels, i.e. each tile consists of *length* times *length* pixels. As a result of this, X and Y must be divisible by *length* without remainder, and $x_{Tmax} \cdot length = X$ and $y_{Tmax} \cdot length = Y$. Each tile is located at a fixed position (x_T, y_T). Similar to pixels, the position of the bottom-left tile of the game world shall be denoted as $(0, 0)$, and the position of the top-right tile of the game world shall be (x_{Tmax}, y_{Tmax}).

A game world shall have geological features, including passable terrain (air, water, magma) and impassable terrain (solid ground). It shall also contain game objects such as the player character `Mazub`, enemy characters and collectable items. The position of geological features of the game world shall always be determined by means of the position of a tile (x_T, y_T) bearing the feature. The feature then affects all pixels belonging to that tile. If a tile of the game world is not assigned a feature explicitly, "air" should be used as the default. Game objects are typically rectangular and occupy a number of pixels. The position of game objects shall always be determined by the position (x, y) of the pixel that is occupied by the bottom-left pixel of the game object. The presence of game objects in a tile only affects those pixels that are actually occupied by the game object. Game objects may occupy any pixel of passable terrain tiles. Game objects may only occupy the top-most row of pixels of solid terrain tiles. All numerical aspects of the game world and positions shall be worked out using integer numbers.

Before the start of a new game, a game world is created so that it contains at least one player character `Mazub` and no more than 100 other game objects. The game world may at all times contain no objects other than this player character and the game objects. The initial positions of `Mazub` and all game objects is passed explicitly to those game objects at the time of creation. Game objects are removed from the game world if their bottom-left position (x, y) leaves the boundaries $(0, 0)..(x_{max}, y_{max})$ of the game world. Further conditions for the "death" of a game object may be specified for each of the different classes below. If a game object's death conditions are met while the object is still located within the boundaries of the game world, the death game object shall be removed from the game world with a delay of $0.6s$ of game time. During this time the game object is not moving but may still passively interact with other game objects. The game terminates when `Mazub` is removed from the game world or reaches a designated target tile.

To account for terrain features of the game world, your implementations of `World`, `Mazub` and other game objects aim to ensure that all pixels of the game world that overlap with a game object belong to passable terrain tiles and are not currently occupied by other game objects. An algorithm that performs sufficiently fine-grained collision detection is outlined in Sec. 2. The outer layer of a game object's pixels is explicitly allowed to overlap impassable terrain and game objects. For a rectangular game object at position (x, y) that occupies an area of X_G times Y_G pixels, this perimeter is defined by the left and right side of the game object, comprising of the coordinates $(x, y+1..y+Y_G-2)$ and $(x+X_G-1, y+1..y+Y_G-2)$, and the bottom and top sides of the game object, comprising of the coordinates $(x..x+X_G-1, y)$ and $(x..x+X_G-1, y+Y_G-1)$. Your implementation may employ the left, right, top and bottom perimeters of game objects to determine if a game object in its current position is capable of conducting certain actions, such as moving or jumping. Specific interactions of a game object with overlapping terrain or other game objects may be specified for each game object or action in the following sections.

The class `World` provides methods to inspect the game world's dimensions X and Y and the *length* of its tiles. The class must also provide a method to ask for the feature whose bottom-left pixel is positioned on a given position. That method must return its result in constant time. `World` shall further implement a method `advanceTime` that iteratively invokes `advanceTime` of all game objects inhabiting the world, starting with the player object `Mazub`. No documentation must be worked out for the method `advanceTime`. In addition, students who are doing the project on their own must not work out any documentation for non-public methods of `World`, provided there is a very good reason not to make them public.

The Graphical User Interface (GUI) for **Jumping Alien** will, in most cases, only display a relatively small rectangular window of the game world. The exact dimensions of this window shall be given in game-world pixels and may be different for each game world. Yet, the window shall never be bigger than the game world. The window shall be approximately centered around the player object `Mazub`. More specifically, unless Mazub is located close to the borders of the game world, the window shall always be positioned so that there are at least 200 pixels between all pixels occupied by `Mazub` and the borders of the visible window. Restrictions to the window's position relative to the position of `Mazub` do not apply if the visible window has the same size as the game world. The class `World` shall provide methods to inspect the position (bottom-left corner) and the height and width of the display window.

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. 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 move to the right or left side of the game world, passing through tiles of passable terrain. 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 $v_x = 1\ m/s$ in the given direction, accelerating with $a_x = 0.9\ m/s^2$ in that direction, up to a maximum velocity of $v_{x_{max}} = 3\ m/s$. `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 horizontal velocity of `Mazub` shall drop to zero immediately as `endMove` is invoked. If there are multiple ongoing movements at the same time, e.g. `startMove(left)` has been invoked while `Mazub` was already moving to the right, the horizontal velocity shall not be set to zero before all ongoing movements are terminated.

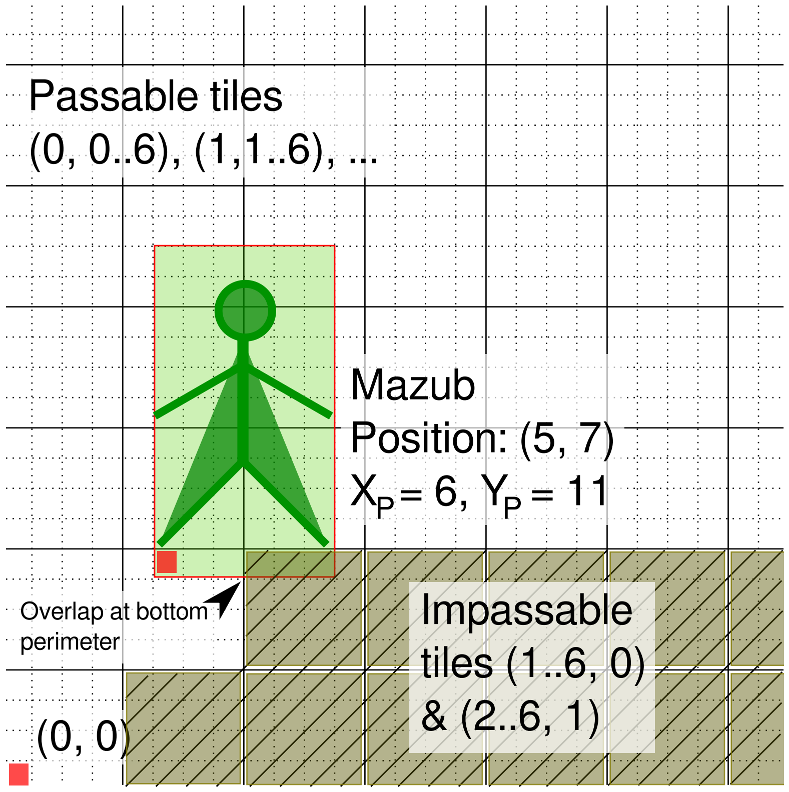

Figure 1: **Jumping Alien**: The game world with pixels, tiles and the player character `Mazub`. `Mazub`is located in passable terrain tiles, standing on the impassable tile $(2, 1)$.

The values for the initial horizontal velocity, the maximum horizontal velocity and the horizontal acceleration may change with respect to interaction with other game objects, terrain and actions performed by `Mazub`. 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, its absolute value will always be greater than zero. 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 0.2 s. 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.

Mazub is moving through passable terrain tiles or on top of solid game world terrain or other game objects, i.e., with only its bottom perimeter overlapping with impassable terrain or other game objects. Mazub shall not move if its side perimeter in the direction of movement overlaps with impassable terrain or another game object.

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.

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 while Mazub is located on top of solid ground or another game object, Mazub starts moving with a velocity of $v_y = 8\ m/s$ 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. Additionally, if Mazub overlaps with impassable terrain or other game objects while Mazub's vertical velocity is greater than zero, Mazub's vertical velocity shall be set to zero, effectively ending the jump.

When Mazub's bottom perimeter does not overlap with pixels belonging to impassable tiles or game objects, Mazub shall *fall*. More specifically, Mazub shall accelerate with $a_y = -10\ m/s^2$ in positive y-direction until Mazub's bottom perimeter reaches the top-most row of pixels of an impassable tile, game object, or leaves the map. As Mazub stands on impassable terrain or another game object, a_y shall be set to 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 $y_{new} = y_{current} + v_{y_{current}} \Delta t + \frac{1}{2} a_y \Delta t^2$. As for running, 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, which is 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 $1m/s$. For your implementation you may safely assume that Y_P for a ducking `Mazub` is smaller than Y_P for a `Mazub` who is not ducking. If `endDuck` is invoked in a location where the appropriate non-ducking Y_P would result in `Mazub` overlapping with impassable terrain, `Mazub` shall continue to duck until appropriate space is available. Thus, `startDuck` may be invoked while `Mazub` is still trying to stand up from previously ducking.

1.2.4 Character Size and Animation

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.

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 $n \geq 10$ 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 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 $\texttt{images}_i$. 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`.

1.2.5 Hit-Points, Metabolism and Enemy Interaction

`Mazub` is assigned number of hit-points. All numerical aspects related to these hit-points shall be worked out using integer numbers and total programming. At the beginning of a game, `Mazub` is assigned 100 hit-points. The current number of hit-points may change during the game as a response to actions performed by `Mazub`. It shall, however, never be lower than 0, indicating `Mazub`'s death, and never be greater than 500.

`Mazub` can gain hit-points by consuming Alien `Plant` objects. As any of

`Mazub`'s perimeters overlap with a `Plant` object while `Mazub` has less than 500 hit-points, `Mazub`'s hit-points shall be increased by 50 and the `Plant` object shall be removed from the game world. Otherwise the `Plant` is not affected by the contact.

`Mazub` can lose hit-points due to contact with enemy objects, i.e., `Slime` and `Shark`. As any of `Mazub`Mazub's perimeters other than the bottom perimeter overlap with a `Slime` or a `Shark`, `Mazub`'s hit points shall be decreased by 50. After losing hit-points due to contact with a `Slime` or `Shark`, subsequent interactions between `Mazub` and enemy objects shall have no effect other than blocking movement (as specified above) for 0.6 s. Properties of the ongoing movement of the colliding game, e.g. direction, velocity and acceleration, may not change directly as a result of the collision.

`Mazub` can further lose hit-points due to contact with water or magma. As long as any of `Mazub`'s perimeters overlap with a terrain tile containing water, `Mazub`'s hit-points shall be decreased by 2 per 0.2 s. If `Mazub` remains in contact with water for less than 0.2 s, no hit-points shall be deduced. As long as any of `Mazub`'s perimeters overlap with a terrain tile containing magma, `Mazub`'s hit-points shall be decreased by 50 per 0.2 s. Any contact with magma shall immediately incur the loss of hit-points but no more than 50 hit-points shall be deduced per 0.2 s.

The class `Mazub` shall provide a method to inspect the current number of hit-points.

1.2.6 Death

`Mazub` dies and is to be removed from the game world as its hit-points drop to zero or below or when its bottom-left pixel, i.e., the position pixel, leaves the boundaries of the game world.

1.3 Other Game Objects

The following sections describe the behaviour of game objects other than `Mazub`, i.e., the classes `Plant`, `Slime` and `Shark`. As these classes bear a lot of similarity with the player character, we describe mainly the differences between `Mazub` and these classes below. A key difference between `Mazub` and plants, slime blobs and sharks is that the latter are not controlled by the player. Yet, they do provide a method `advanceTime` that shall randomly select an action from the list of options specified for each class. Each action period is initiated by invoking the method `start<Action>` and finishes with invoking the corresponding `stop<Action>` method of the game object. The movement of all game objects mostly adheres to the same rules: movement is blocked by impassable terrain and other game objects. Contact with other game objects or terrain features has no impact on the duration of random actions performed by a game object. All game objects described below must be created with an array of two sprites that are to be returned by `getCurrentSprite` for movement to the left (default, index 0) and to the right (index 1). All aspects of the classes `Plant`, `Slime` and `Shark` shall be specified only in a formal way.

1.3.1 The Class `Plant`

Alien `Plants` primarily act as food for `Mazub`. They neither jump nor fall nor duck but are capable of hovering on passable terrain. `Plants` possess one hit-point and are destroyed upon contact with a hungry `Mazub`. Contact with other game objects does not affect `Plants` and they also do not lose hit-points when making contact with water or magma. Until destruction, `Plants` shall move to the left and right with a constant horizontal velocity of 0.5 m/s for 0.5 s of game time, alternating.

1.3.2 The Class `Slime`

`Slimes` must not be worked out by students that are doing the project individually. `Slime` blobs are pretty dumb predatory land beasts aiming to devour `Mazub`. They possess all properties of `Mazub` except for the abilities to duck, jump and to consume Alien `Plants`. They start the game with 100 hit-points and move randomly to the left or right, accelerating with an $a_x = 0.7\ m/s^2$ up to a maximum velocity of $v_{x_{max}} = 2.5\ m/s$. Each movement period shall have a duration of 2 s to 6 s. `Slime` blobs lose 50 hit-points when making contact with `Mazub` or `Sharks`. While they are non-hostile to each other, i.e., they do not lose hit-points on contact with another `Slime`, they do block each others' movement. `Plants` shall not block the movement of Slime. As Mazub, Slime objects lose hit-points upon touching water or magma.

Slimes are always organised in groups, called schools. In particular, a living Slime belongs at all times to exactly one school. This does not prevent Slimes from switching from one school to another. Each time a Slime looses some hit-points, all other Slimes in the same school will loose 1 hit-point. At the time a Slime joins another school, it will hand over 1 hit-point to all existing members of the old school. At the same time, all existing members of the new group will hand over 1 hit-point to the new member. If a Slime collides with another Slime of a different school, the Slime of the smaller school shall join the larger school. If the schools are equally large, both Slimes remain in their original schools. At all times there shall be no more than 10 schools of slime in a game world.

1.3.3 The Class Shark

Sharks differ from Slime in their abilities: they are capable of jumping. They also differ from Mazub as they cannot duck and do not interact with Plants.

Sharks start the game with 100 hit-points. They typically appear in water tiles and do not lose hit-points while submerged in water. Yet, they do lose 6 hit-points per 0.2 s while in contact with air. If a Shark remains in contact with air for less than 0.2 s, no hit-points shall be deduced. As Mazub and Slime blobs, Sharks lose hit-points upon touching magma.

Sharks are capable of jumping while their bottom perimeter is overlapping with water or impassable terrain. Specifically, they may jump while moving to the left or right, accelerating with an $a_x = 1.5\ m/s^2$ up to a maximum velocity of $v_{x_{max}} = 4\ m/s$. Sharks' initial velocity for jumping shall be set to $v_y = 2\ m/s$. Each movement period of a Shark must have a duration of 1 s to 4 s. A jump shall occur at the start of a horizontal movement period, and the shark stops jumping at the end of that period (this interferes neither with a premature end of the jump due to collisions, nor with an extended period falling after the jump). There must be at least four non-jumping periods of random movement in between the end of one jump and the start of the next one.

Similar to `Mazub` and `Slime` blobs, sharks fall while their bottom perimeter is not overlapping with impassable terrain or other game objects. Additionally, as they are capable of swimming, they stop falling as soon as they are submerged in water, i.e a falling `Shark`'s vertical velocity and vertical acceleration shall be set to zero once the `Shark`'s top perimeter is overlapping with a water tile. While submerged in water, `Sharks` are capable of diving and rising: each non-jumping period of horizontal movement shall be extended by a random vertically acceleration of $-0.2\ m/s^2 \leq a_y \leq 0.2\ m/s^2$. The vertical acceleration of a non-jumping `Shark` shall be set to zero if the `Shark`'s top or bottom perimeters are not overlapping with a water tile any more, and at the end of the movement period.

`Sharks` lose 50 hit-points when making contact with `Mazub` or `Slime` blobs. They are non-hostile to each other but block each others' movement. `Plants` shall not block the movement of `Sharks`.

2 Collision Detection

This section describes an idea for a simple algorithm to detect collisions between game objects, and game objects and terrain features. The focus of the approach presented here is on efficiency and on achieving a high enough precision for the purpose of your game implementation. The algorithm does, by no means, aim at realistically modelling physics and you are free to opt for another algorithm in your implementation of the functional requirements of **Jumping Alien**.

Our algorithm relies strongly on features of the game world described above: all geological features and game objects are rectangular in shape and in-game-positions are based on square pixel coordinates. Furthermore, game objects move in isolation, one after another, while all other game objects do not move. Fig. 2 illustrates this setup: `Mazub` positioned at (x, y) and `advanceTime` is invoked with some Δt that, given `Mazub`'s current velocity and acceleration would result in `Mazub` moving to (x', y'). However, we wish to detect if `Mazub` overlaps with any of the terrain features or game objects (blue rectangles) on its way from (x, y) to (x', y').

Intuitively our algorithm aims to slice the time advancement Δt into smaller fractions dt of time that ensure that a game object moves approximately 1 *cm*, the side length of a pixel, per time slice. This allows us to split the entire movement into fractional movements from one pixel to another, followed by immediate checks for overlapping of the moving game object with the surrounding terrain features and other game objects. Indeed, after moving for seven of these time slices, we see that `Mazub` overlaps with one of the blue rectangles (overlapping pixels marked in dark blue) and take actions as specified in the previous section.

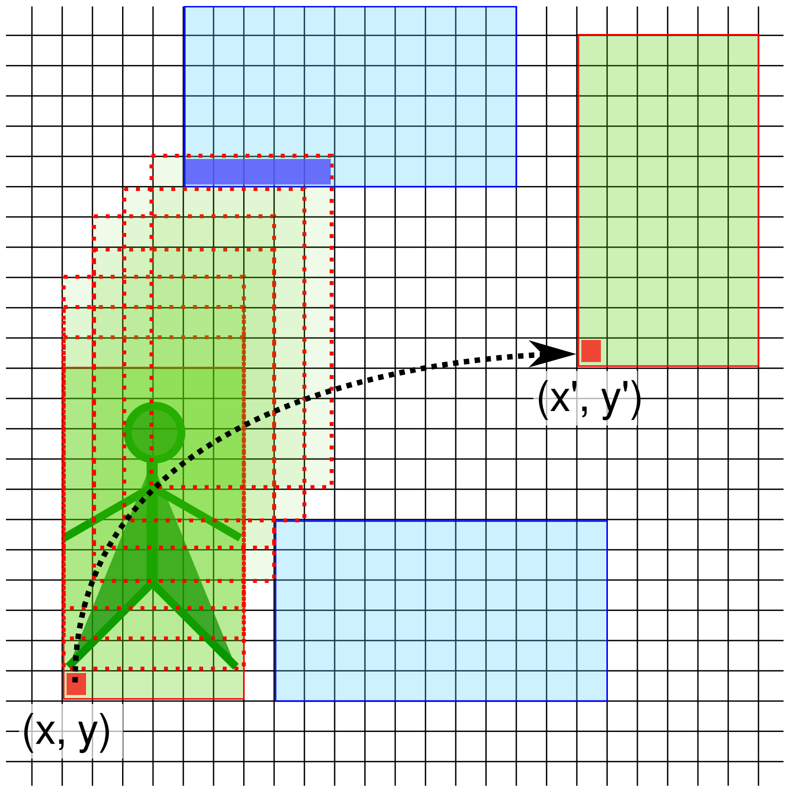

Figure 2: **Jumping Alien**: Collision detection.

As can be seen, choosing a reasonably small dt is key for achieving acceptable performance and precision. We propose to determine dt as the time needed to travel 0.01 m at the current velocity of the game object that is to be moved. Specifically, we compute dt as follows:

$$dt = \min \left(\begin{array}{r} \frac{1}{|\frac{v_x}{100}|}, \\ \frac{1}{|\frac{v_y}{100}|}, \\ \frac{\sqrt{2|\frac{a_x}{100}|+\frac{v_x}{100}^2}-|\frac{v_x}{100}|}{|\frac{a_x}{100}|} \text{ if } a_x \neq 0, \\ \frac{\sqrt{2|\frac{a_y}{100}|+\frac{v_y}{100}^2}-|\frac{v_y}{100}|}{|\frac{a_y}{100}|} \text{ if } a_y \neq 0 \end{array} \right)$$

Now we iteratively advance time for dt (using the effective velocities *and* acceleration) and compute the in-game-position of the moved game object. Based on the in-game-position we can determine if some object $((x, y), X_P, Y_P)$ does not overlap with another object $((x', y'), X'_P, Y'_P)$: the two objects do not overlap if $(x + (X_P - 1) < x' \vee x' + (X'_P - 1) < x \vee y + (Y_P - 1) < y' \vee y' + (Y'_P - 1) < y)$; otherwise they do overlap.

Optionally you may choose to optimise the algorithm by determining a set of likely candidates for terrain features and game objects to overlap with the moving object, rather than repeatedly checking for intersection with all terrain and game objects.

As other strategies to detect collisions are allowed, methods for collision detection shall not reveal in their documentation any internal details concerning he actual strategy. This does not mean that a method such as `collidesWith(other)` must reveal in its documentation when "this" collides with "other".

3 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/.

4 Testing

Write JUnit test suite for the classes specified in this document, that tests each public method. Include this test suite in your submission.

5 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.part2.facade` that implements `IFacadePart2` (which extends `IFacade` from part 1). `IFacade.java` and `IFacadePart2.java` contain 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 `JumpingAlienPart2`. After starting the program, you can press keys to modify the state of the program. Commands to `Mazub` 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 focus of this assignment is with respect to classes specified in this document. 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.

6 Submitting

The solution must be submitted via Toledo as a JAR file individually by all team members before the 27th of April 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!

There will be no feedback session to discuss this part of the project. Your solution for this part will be evaluated during the final defense. You are free to change your solution for this part while working out the final part of the project. The assignment for that part will be available around the 20th of April.