OGP Assignment 2017-2018:
Worms (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; only in exceptional situations the assignment can be made 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. 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.

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. 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 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 the *Git* version control system. Instructions on how to obtain a private repository on GitHub, already populated with the provided GUI code (see section 4), as well as a short tutorial, will appear in a separate

document on Toledo.

During the assignment, we will create a simple game that is loosely based on the artillery strategy game *Worms*. Note that several aspects of the assignment will not correspond to the original game. Your solution should be implemented in Java 8 or higher and follow the rules described in this document.

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

Worms is a turn-based artillery strategy game in which the player controls a team of worms that can move in a two-dimensional landscape. The worms are equipped with tools and weapons that are to be used to achieve the goal of the game: kill the worms of other teams and have the last surviving worms. In this assignment, we will create a game loosely based on the original artillery strategy released in 1995 by Team17 Digital.

In the first part of the assignment, we focus on a single class `Worm`. However, your solution may contain additional helper classes (in particular classes marked *@Value*). In the second and third part, we will add additional classes to our game. In the remainder of this section, we describe the class `Worm` in more detail. All aspects of your implementation must be specified both formally and informally.

1.1 Properties of Worms

Each worm is located at a certain location (x, y) in a two-dimensional space. Both x and y are expressed in metres (m). At this moment, the two-dimensional space is unbounded in both directions, meaning that worms can be located at infinity. All aspects related to the location of a worm shall be worked out *defensively*.

Each worm faces a certain direction. The orientation of a worm is expressed as an angle θ in radians. For example, the orientation of a worm facing right is 0, a worm facing up has $\pi/2$ as its orientation, a worm facing

left has an orientation equal to π and a worm facing down has $3\pi/2$ as its orientation. The orientation of a worm will always be in the range $0 \ldots 2\pi$, the latter value not included. All aspects related to the orientation must be worked out *nominally*.

The shape of a worm is a circle with finite radius σ (expressed in metres) centred on the worm's location. The radius of a worm must at all times be at least 0.25 m. Yet, the effective radius of a worm may change during the program's execution. In the future, the lower bound on the radius may change and it is possible that different lower bounds will then apply to different worms. However, the lower bound for a single worm will never change during the lifetime of that worm. All aspects related to a worm's radius must be worked out *defensively*.

Each worm also has a mass m expressed in kilograms (kg). m is derived from σ, assuming that the worm has a spherical body and a homogeneous density p of 1062 kg/m^3: $m = p \cdot (4/3 \cdot \pi\sigma^3)$.

Each worm has a maximum number of action points, and a current number of action points, which shall be represented by integer values. The maximum number of action points of a worm must be equal to the worm's mass m, rounded to the nearest integer using the predefined method `round` in `java.lang.Math`. If the mass of a worm changes, the maximum number of action points must be adjusted accordingly. The current number of action points may change during the program's execution. Yet, the current value of a worm's action points must always be less than or equal to the maximum value, but it must never be less than zero. Whenever a worm is created, its current number of action points will have the maximum value. All aspects related to action points must be worked out in a total manner.

If not stated otherwise, all numeric characteristics of a worm shall be treated as double precision floating-point numbers. That is, use Java's primitive type `double` to store the radius, the x-coordinate, etc. The characteristics of a worm must be valid numbers (meaning that `Double.isNaN` returns `false`) at all times. However, we do not explicitly exclude the values `Double.NEGATIVE_INFINITY` and `Double.POSITIVE_INFINITY` (unless specified otherwise).

In addition to the above characteristics, each worm shall have a name. A worm's name may change during the program's execution. Each name is at least two characters long and must start with an uppercase letter. In the current version, names can only use letters (both uppercase and lowercase), quotes (both single and double) and spaces. James o'Hara is an example of a well-formed name. It is possible that additional characters may be allowed in later versions of the game. All aspects related to the worm's name must be worked out defensively.

The class `Worm` shall provide methods to inspect name, location, orientation, radius, mass, and action points of a worm.

1.2 Turning and Moving

A worm can move and turn. The class `Worm` shall provide a method `move` to change the location of the worm based on the current location, orientation, and a number of steps. Movement always occurs in steps. The distance covered in one step shall be equal to the radius of the worm. The given number of steps shall never be less than zero. As this method affects the location of the worm, it must be worked out defensively.

The class `Worm` must provide a method `turn` to change the orientation of the worm by adding a given angle to the current orientation. As this method affects the orientation, it must be worked out nominally. This means that the given angle must be such that the resulting angle is in the specified range for the orientation of a worm.

Active turning and moving costs action points. Changing the orientation of a worm by 2π shall decrease the current number of action points by 60. Respectively, changing the orientation of a worm by a fraction of $2\pi/f$ must imply a cost of $60/f$ action points. The cost of movement shall be proportional to the horizontal and vertical component of the step such that a horizontal step is at the expense of 1 action point, while a vertical step incurs costs of 4 action points. The total cost of a step by a worm with orientation θ can be computed as $|\cos\theta| + |4\sin\theta|$. Since action points are to be handled as integer values, all expenses of action points shall be rounded up to the next integer.

1.3 Jumping

Worms may also jump along ballistic trajectories. The class `Worm` shall provide a method `jump` to change the location of the worm as the result of a jump from the current location (x, y) and with respect to the worm's orientation θ and the number of remaining action points APs. All methods related to jumping must be worked out defensively.

Given the remaining activity points APs and the mass m of a worm, the worm will jump off by exerting a force of $F = (5 \cdot APs) + (m \cdot g)$ for 0.5 s on its body. Here, g represents the standard acceleration in the game world which is equal to 5.0 m/s^2. [1] From this, we can compute the initial

[1] We use a fictitious value for the standard acceleration instead of the standard acceleration on earth to make testing simpler.

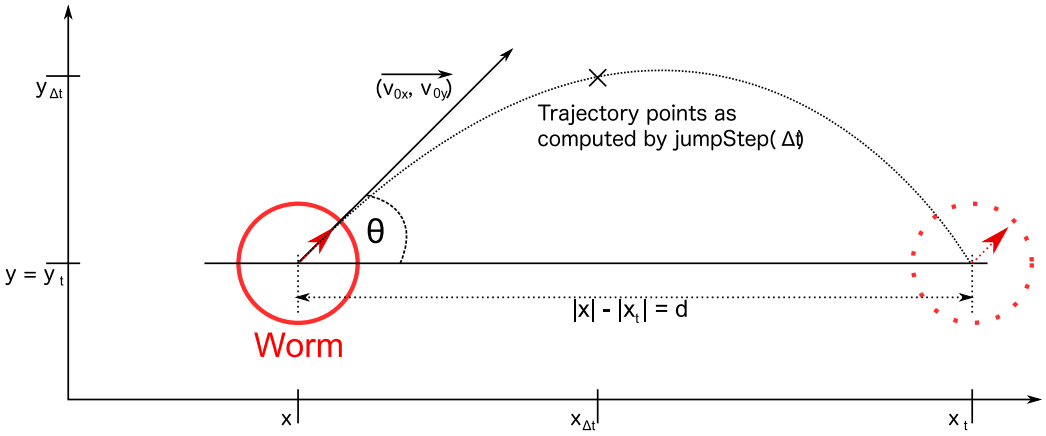

Figure 1: Illustration of a jumping worm's trajectory.

velocity of the worm as $v_0 = (F/m) \cdot 0.5\ s$. The formula to calculate the initial velocity may change in the future. However, the resulting value will always be nonnegative and finite.

The class `Worm` shall provide a method `jumpStep` that computes in-flight locations $(x_{\Delta t}, y_{\Delta t})$ of a jumping worm at any Δt seconds after launch. $(x_{\Delta t}, y_{\Delta t})$ may be computed as follows:

$$\begin{aligned} v_{0x} &= v_0 \cdot cos\,\theta \\ v_{0y} &= v_0 \cdot sin\,\theta \\ x_{\Delta t} &= x + (v_{0x}\Delta t) \\ y_{\Delta t} &= y + (v_{0y}\Delta t - \tfrac{1}{2}g\Delta t^2) \end{aligned}$$

As illustrated in Fig. 1.3, jumping worms always travel horizontally as if launched from a solid ground line parallel to the x-axis at the worm's y location, and return to that line. This means that a worm will jump a distance $d = ({v_0}^2 \cdot sin(2\theta))/g$ horizontally, within the following $t = d/(v_0 \cdot cos\,\theta)$ seconds. The class `Worm` shall provide a method `jumpTime` that returns the above t for a potential jump from the current location, If the worms orientation is in the range $\pi < \theta < 2\pi$, i.e. the worm is facing downwards, the worm shall not move. Jumping consumes all remaining action points of a worm.

The methods `jumpTime` and `jumpStep` must not change any attributes of a worm. The above equations represent a simplified model of terrestrial physics and consider uniform gravity with neither drag nor wind. Future phases of the assignment may involve further trajectory parameters or geographical features of game world.

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[2]

$$0.2000000000000000011102230246251565404236316680908203125$$

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

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

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

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

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 Testing

Write JUnit test suite for the class `Worm` with tests for the methods to move, to turn and to jump. 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 worms. The user interface is included in the assignment as a JAR file. When importing this JAR file into Eclipse as an existing project, you will find a folder `src-provided` that contains the source code of the user interface 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 `worms.facade` that implements the provided interface `IFacade` from package `worms.facade`. `IFacade.java` contains additional instructions on how to implement the required methods. Read this documentation carefully.

To start the program, run the `main` method in the class `worms.Worms`. After starting the program, you can press keys to modify the state of the program. The command keys are [Tab ⟶] for switching worms, [←] and [→] (followed by pressing [⏎]) to turn, [↑] to move forward, [+] and [−] to increase and decrease the worm's radius, [n] to change the worm's name, [j] to jump, and [esc] to terminate the program. Be aware that the GUI displays only part of the (infinite) space. Your worms may leave and return to the visible area.

You can freely modify the GUI as you see fit. However, the main focus of this assignment is the class `Worm`. 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 11th of March 2018 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 19th and the 30st of March. More information will be provided via Toledo.