

# OGP Assignment 2016-2017: Asteroids (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 based on this assignment. The assignment is preferably taken in groups of two students; only in exceptional situations the assignment can be worked out 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](mailto:ogp-inschrijven@cs.kuleuven.be) **before the 1st of March 2017**. If you cooperate, only one team member shall send this 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](mailto:ogp-inschrijven@cs.kuleuven.be) and each of the group members is then required to complete the project on their own.

During the assignment, we will create a simple game loosely based on the arcade game *Asteroids*. Note that several aspects of the assignment will not correspond to the original game. 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 the specified 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 neither impose to use nominal programming, total programming, nor 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 underlying concepts of object-oriented programming. Specifically, the goal of this exercise is not to hand in the best possible arcade game. Therefore, your grades do not depend on correctly implementing functional requirements only; we will pay attention to documentation, accurate specifications, re-usability and adaptability as well. After handing in your solution to the first part of the assignment, you will receive feedback on your submission. After handing in 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 a 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

*Asteroids* is an arcade game where the player controls a spacecraft in an asteroid field. The goal of the game is to evade and destroy objects such as enemy vessels and asteroids in a two-dimensional, rectangular space. In this assignment, we will create a game loosely based on the original arcade game released in 1979 by Atari.

In the first part of the assignment, we focus on a single class `Ship`. 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 `Ship` in more detail. All aspects of your implementation shall be specified both formally and informally.

### 1.1 Position, Velocity, Orientation and Radius

Each spaceship is located at a certain position  $(x, y)$  in an unbounded two-dimensional space. Both  $x$  and  $y$  are expressed in kilometres ( $km$ ). All

aspects related to the position of a ship shall be worked out **defensively**.

Each spaceship has velocities  $v_x$  and  $v_y$  that determine the vessel's movement per time unit in the  $x$  and  $y$  direction, respectively. Both  $v_x$  and  $v_y$  are expressed in kilometres per second ( $km/s$ ). The speed of a ship, computed as  $\sqrt{v_x^2 + v_y^2}$ , shall never exceed the speed of light  $c$ ,  $300000km/s$ . In the future, this limit need not remain the same for each ship, but it will always be less than or equal to  $c$ . All aspects related to velocity must be worked out in a **total** manner.

Each ship has an orientation, i.e., it faces a certain direction expressed as an angle in radians. For example, the orientation of a ship facing right is 0, a ship facing up is at angle  $\pi/2$ , a ship facing left is at angle  $\pi$  and a ship facing down is at angle  $3\pi/2$ . The orientation of a ship must always be a value in between 0 and  $2\pi$ . All aspects related to the orientation must be worked out **nominally**.

The shape of each ship is a circle with radius  $\sigma$  (expressed in kilometres) centred on the ship's position. The radius of a ship must be larger than 10 km. It never changes during the program's execution. In the future, this lower bound may change, however it will always remain the same for each ship and its value will be positive. All aspects related to the radius must be worked out **defensively**.

Conceptually, all of the above characteristics of a ship are real numbers.

The class **Ship** shall provide methods to inspect the position, velocity, orientation and radius.

## 1.2 Moving, Turning and Accelerating

A spaceship can move, turn and accelerate. The class **Ship** shall provide a method **move** to change the position of the spacecraft based on the current position, velocity and a given time duration  $\Delta t$ . The given duration  $\Delta t$  shall never be less than zero. If the given duration is zero or the ship's velocity is zero, the ship shall keep its current position. As this method affects the position of the ship, it must be worked out **defensively**.

The class **Ship** must provide a method to turn the ship by adding a given angle to the current orientation. This method must be worked out **nominally**.

Finally, **Ship** must provide a method **thrust** to change the ship's velocity based on the current velocity  $v$ , its orientation  $\theta$ , and on a given amount  $a$ . The new velocity of the ship  $(v'_x, v'_y)$  is derived as follows:

$$\begin{aligned}v'_x &= v_x + a \cdot \cos(\theta) \\v'_y &= v_y + a \cdot \sin(\theta)\end{aligned}$$

The given amount  $a$  must never be less than zero. This method must be

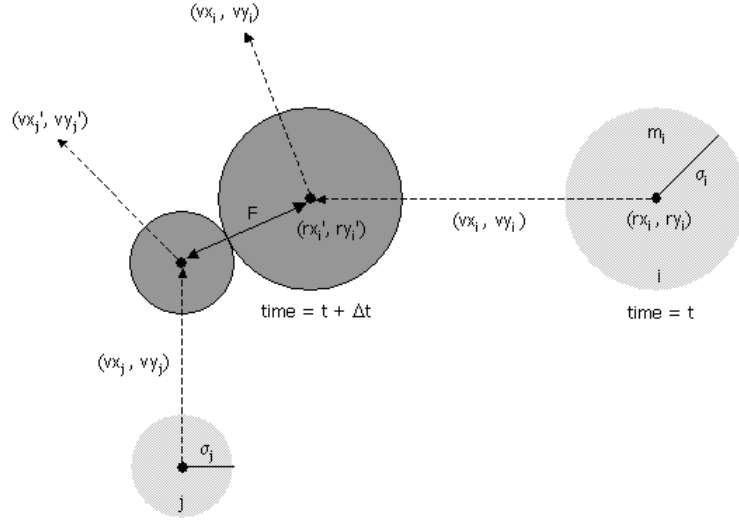
worked out totally (replacing  $a$  by zero, if it is less than zero). If the new velocity's magnitude would exceed the upper bound  $c$ , then reduce  $v_x$  and  $v_y$  such that the speed becomes  $c$  (but do not modify the new direction of the velocity). For simplicity, we assume that turning and accelerating does not take any time.

### 1.3 Collision Prediction

Spacecrafts may collide with each other and the next phases of the assignment will specify the collision behaviour. For now the vessels may pass each other and overlap without altering their properties. Yet, you shall add functionality to detect overlapping and to predict collisions. To do so, **Ship** shall also provide the following methods:

- **getDistanceBetween** returns the distance in between two spacecrafts. The distance may be negative if both ships overlap. The distance between a ship and itself is zero.
- **overlap** returns **true** if and only if two spacecrafts overlap. A ship always overlaps with itself.
- **getTimeToCollision** shall return when (i.e. in how many seconds), if ever, two spacecrafts will collide. **getTimeToCollision** shall return **Double.POSITIVE\_INFINITY** if the ships never collide. This method does not apply to ships that overlap.
- **getCollisionPosition** shall return where, if ever, two spacecrafts will collide. The method shall return **null** if the ships never collide. This method does not apply to ships that overlap.

Implement these methods **defensively**. Below we explain how collision points and collision times can be computed.



- Given the positions and velocities of two spacecraft  $i$  and  $j$  at time  $t$ , we wish to determine if and when they will collide with each other.
- Let  $(rx'_i, ry'_i)$  and  $(rx'_j, ry'_j)$  denote the positions of spacecraft  $i$  and  $j$  at the moment of contact, say  $t + \Delta t$ . When the spacecrafts collide, their centres are separated by a distance of  $\sigma = \sigma_i + \sigma_j$ . In other words:  

$$\sigma^2 = (rx'_i - rx'_j)^2 + (ry'_i - ry'_j)^2$$
- During the time prior to the collision, the spacecrafts move in straight-line trajectories. Thus,  

$$rx'_i = rx_i + \Delta t \cdot vx_i, ry'_i = ry_i + \Delta t \cdot vy_i$$

$$rx'_j = rx_j + \Delta t \cdot vx_j, ry'_j = ry_j + \Delta t \cdot vy_j$$
- Substituting these four equations into the previous one, solving the resulting quadratic equation for  $\Delta t$ , selecting the physically relevant root, and simplifying, we obtain an expression for  $\Delta t$  in terms of the known positions, velocities, and radii:

$$\Delta t = \begin{cases} \infty & \text{if } \Delta v \cdot \Delta r \geq 0, \\ \infty & \text{if } d \leq 0, \\ -\frac{\Delta v \cdot \Delta r + \sqrt{d}}{\Delta v \cdot \Delta v} & \text{otherwise,} \end{cases}$$

where  $d = (\Delta v \cdot \Delta r)^2 - (\Delta v \cdot \Delta v)(\Delta r \cdot \Delta r - \sigma^2)$  and

$$\Delta r = (\Delta x, \Delta y) = (rx_j - rx_i, ry_j - ry_i)$$

$$\Delta v = (\Delta vx, \Delta vy) = (vx_j - vx_i, vy_j - vy_i)$$

$$\Delta r \cdot \Delta r = (\Delta x)^2 + (\Delta y)^2$$

$$\Delta v \cdot \Delta v = (\Delta vx)^2 + (\Delta vy)^2$$

$$\Delta v \cdot \Delta r = (\Delta vx)(\Delta x) + (\Delta vy)(\Delta y).$$

We expect a declarative specification of the method `getTimeToCollision`. This means that you must specify conditions to be satisfied by the time returned by the method instead of repeating the implementation as part of the specification. In other words, your specification should not be a mere copy of the body of the method. You must only work out a specification for the case in which the method returns a finite value. You lose 0.5 points on a total of 3.0 if your specification does not satisfy the above criteria.

## 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<sup>1</sup>

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:

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

---

<sup>1</sup>You can check this by running `System.out.println(new BigDecimal(1.0/5.0))`.

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 Testing

Write a JUnit test suite for the class **Ship** 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 spaceships. 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 **asteroids.facade** that implements the provided interface **IFacade** from package **asteroids.part1.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 of the provided class **Part1** in package **asteroids.part1**. After starting the program, you can press keys to modify the state of the program. The command keys are  for switching spacecrafts,  and  to turn,  to accelerate,  to show

collision points, and `Esc` to terminate the program. Be aware that the GUI displays only part of the (infinite) space. Your spacecrafts may exit and re-enter the visible area.

You can freely modify the GUI as you see fit. However, the main focus of this assignment is the class `Ship`. 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 **12th of March 2017 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 20th and the 31st of March 2017**. More information will be provided via Toledo.