

OGP Assignment 2016-2017: asteroids (Part III)

This text describes the [third](#) part of the assignment for the course *Object-oriented Programming* (OGP). There is no exam for this course. [If you worked out part two of the project with some partner, you must work out this final part of the project with the same partner. You are not allowed to start working with a new partner for part three. As before, if conflicts arise within the team, you must report them to \[ogp-inschrijven@cs.kuleuven.be\]\(mailto:ogp-inschrijven@cs.kuleuven.be\) before the 10th of May 2017. Both members of the team must then finish the project on their own. There are some reductions in the third part of the assignment that apply to all students that are working individually on the project.](#)

During the assignment, we will create a simple game loosely based on the arcade game *asteroid*. 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. [On Toledo you find a document listing minimal requirements to pass for this course, requirements to get a score of 14 or more, and requirements to get a score of 17 or more. The assignment itself also includes some aspects that you must only work out if you want to get a high score for this course.](#)

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 ship in an asteroid field. The goal of the game is to evade and destroy objects such as enemy vessels and minor planets 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 focused on a single class `Ship`. In the second part, we extended `Ship`, and we added game `Worlds` and `Bullets`. [In the third part, we further extend the game](#) (changes marked in blue) [with two kinds of `MinorPlanets`, namely `Asteroids` and `Planetoids`. Moreover, `Ships` are upgraded to entities that can execute programs in a simple domain-specific programming language.](#) As before, your solution may contain additional helper classes (in particular classes marked *@Value*). If the assignment does not specify how certain aspects of the game are to be worked

out, you may select the option you prefer. You may also use inheritance as you see fit.

In the remainder of this section, we describe the classes `Ship`, `Bullet`, `MinorPlanet`, and `World` in more detail. We also describe the programming language to steer ships, and the way programs in that language are to be executed. Unless specified otherwise, all aspects of your implementation of `Ship` shall be documented both formally and informally. The classes `Bullet`, `MinorPlanet` and `World` must be documented in a formal way only. Classes related to programs must not be documented at all.

If the formal documentation of a method is basically the same as its implementation, you may replace the documentation with a simple reference to the implementation. For example:

```
/*
 * Return the Body Mass Index of this person.
 * @see implementation
 */
public getBMI() {
    return this.getWeight()/Math.pow(this.getLength(),2);
}
```

Be careful: If a more declarative specification of a method is possible, we expect that documentation and not a copy of the method's implementation.

1.1 Game World – The Class `World`

A game world is a two-dimensional, rectangular area containing ships and bullets.

Each world has a particular size, described by a width and height expressed in kilometres (*km*). The size of a world cannot change after construction. Both the width and height must lie in the range 0 to `Double.MAX_VALUE` (both inclusive), for all worlds. In the future, the upper bound on the width and height may decrease. However, all worlds will share the same upper bounds. All methods involving the size of a world must be worked out in a **total** way. The world defines a two-dimensional coordinate space with $x \in [0, \text{width}]$ and $y \in [0, \text{height}]$.

A world contains ships, `minor planets` and bullets. At all times, each ship, `minor planet` or bullet is located in at most one world. No world contains the same ship, `the same minor planet` or bullet twice. Ships, `minor planets` and bullets are circular entities. If such an entity is located in a world, then the circle must lie fully within the bounds of that world and it shall not overlap with other ships, `minor planets` or bullets within that world. Exceptions to this rule involve the interaction of bullets with the ship that has fired these bullets, which are explained in the following sections.

To account for rounding issues with the **double**-representation of real numbers in Java (cf. Sec. 2), your implementation of minor planets shall employ a notion of *significant overlap* between two objects whenever checks for overlap of two objects are required: Two objects A and B overlap significantly if the distance between the centres of A and B is $\leq 99\%$ of the sum of the objects' radii σ_A and σ_B .

Similarly, your implementation shall employ a notion of *apparently within boundaries* of an object inside a container whenever checks are needed to verify that the contained object is fully within the boundaries of its containing object. An object A apparently lies within the boundaries of a container C if the distance between each boundary of C and the centre of A is $\geq 99\%$ of the radius of A .

The class `World` shall provide methods for adding and removing ships, **minor planets** and bullets. Those methods must be worked out **defensively**.

The class `World` shall also provide a method to return the ship, **the minor planet** or the bullet, if any, at a given position, i.e. the object whose centre coincides with the given position. This method must return its result in nearly constant time and must be worked out in a **total** way.

Finally, it must be possible to query a game world for the set of all ships, **the set of all minor planets**, the set of all bullets as well as the set of all entities (i.e. all ships, **minor planets** and all bullets) in a given world.



Ideally, the introduction of additional game elements such as minor planets, should be possible without any change to the class `World`. If this is not the case, we strongly advise you to restructure your code first, such that it indeed becomes possible to extend the game with other game elements without any impact on the class `World`.

Hint: Have a look at how no changes are needed in the class of persons when new types of possessions such as jewels and cars become part of the kind of things people can own [Exercise on dogs and paintings].

1.1.1 Advancing Time

The state of a world evolves as time passes. For example, the position of an entity changes over time if the object is flying through space at a non-zero velocity. Similarly, an entity's velocity can change because of collisions.

The class `World` should provide a method `evolve`, which advances the state of the world a certain number of seconds Δt . **You must not work out a specification for this method.** Implement this method in a **defensive** way as follows:

1. Predict the first collision C (i.e. the collision that happens before all other collisions, cf. Sec. 1.5).

2. Suppose C happens in t_C seconds. If t_C is larger than Δt , go to step 5. Otherwise, advance all ships, [minor planets](#) and bullets t_C seconds (to the time right before the first collision).
3. Resolve C (cf. Sec. 1.5.2)
4. Subtract t_C from Δt and go to step 1.
5. Advance all ships, [minor planets](#) and bullets Δt seconds, as explained in the following sections.

Advancing a ship, [minor planet](#) or bullet entails updating its position based on its current velocity. If a ship's thruster is enabled, additionally modify its velocity (*after* updating its position in steps 2 and 5 above) based on its acceleration. Section 1.2 explains how the new velocity of the ship (v'_x, v'_y) is derived.

1.2 The Class Ship

1.2.1 Position, Velocity, Orientation and Radius

Each spaceship is located at a certain position (x, y) . A ship may be associated with a specific **World** under the constraints specified in Sec. 1.1. If a ship is not associated with a world, it is positioned 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 π 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 2π . The class **Ship** must provide a method to turn the ship by adding a given angle to the current orientation. All aspects related to the orientation must be worked out [nominally](#).

The shape of each ship is a circle with radius σ (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**.

Each ship has a mass expressed in kilograms (*kg*). The mass of a ship determines its resistance to acceleration and the effect of collisions with other ships. The mass density ρ of each ship is at least $1.42 \cdot 10^{12} \text{kg}/\text{km}^3$. Furthermore, the mass m of a ship satisfies:

$$m \geq \frac{4}{3}\pi\sigma^3\rho$$

The mass of a ship may change during its lifetime. You may assume that the mass of a ship will always be much smaller than `Double.MAX_VALUE` in Java. In addition, each ship has a total mass which equals the ship's mass plus the mass of objects carried by that ship. All aspects related to a ship's mass and total mass must be worked out **totally**.

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

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

1.2.2 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 ship based on the current position, velocity and a given time duration Δt . The given duration Δt 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**.

Each ship has a thruster. The thruster is either enabled or disabled. When the thruster is enabled, the space ship accelerates in the direction it is facing. Thus, `Ship` must provide the method `thrustOn` and `thrustOff` to enable and respectively disable the thruster. Enabling thrusters shall result in a change the ship's velocity based on the current velocity v , its orientation θ , and on an acceleration a . For now, each ship is equipped with the same kind of thruster. An active thruster exerts a force $F = 1.1 \cdot 10^{18} \text{kg} \cdot \text{km}/\text{s}^2$ on its ship. In future iterations, the value of F may not remain the same for each ship. The acceleration generated by the thruster can be derived from Newton's second law of motion ($F = ma$). The new velocity of the ship (v'_x, v'_y) is derived as follows:

$$\begin{aligned} v'_x &= v_x + a \cdot \cos(\theta) \cdot \Delta t \\ v'_y &= v_y + a \cdot \sin(\theta) \cdot \Delta t \end{aligned}$$



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.

The class **Ship** shall provide methods to inspect the state of the thruster and the ship's acceleration.

1.2.3 Interactions with Bullets

Ships can be hit by bullets, load bullets, and fire bullets (cf. Sec. 1.4 and Sec. 1.5). Similar to ships, **Bullets** are circular entities with a position and a radius.

If a bullet is loaded on a ship, its circle must lie fully within the bounds of that ship. The bullet may, however, overlap with other bullets loaded on that same ship. The class **Ship** must provide methods to extend its collection of bullets with a single bullet as well as with an arbitrary series of bullets.¹ Upon creation, each ship shall not have any bullets loaded.

Whenever a ship fires a bullet, the bullet is removed from the ship's collection of bullets. All methods related to the collection and firing of bullets, and the total number of bullets shall be worked out in a **defensive** way.

Bullets influence the total mass of a ship. The total mass of a ship is equal to the sum of that ship's mass and the mass of all bullets loaded on it.

At all times, ships that are positioned within a world can fire one of their bullets. The initial speed of such a bullet is $250km/s$ and the direction of its velocity is equal to the direction the ship is facing². The bullet is initially placed next to the firing ship and so that the ship and the bullet do not overlap, at the angle the ship is facing. However, if the bullet's initial position is already (partially) occupied by another entity, then the bullet immediately collides with that entity. If the bullet would be located (partially) outside of the ship's world, it is not added to the ship's world but destroyed immediately. Ships that are not positioned in a world cannot fire bullets. The method to fire a bullet must be worked out **totally**.

Students working alone: You may replace the collection of bullets by a counter of the total number of bullets available on the ship. If you add a bullet to a ship or remove a bullet from a ship, you must ensure that the bullet fully overlaps with the ship. If so, the bullet counter must be incremented, respectively decremented, and the bullet shall be destroyed. You are not

¹Use a method with variable arguments for this purpose to load multiple bullets.

²Note that the velocity of the ship itself has no influence on the initial velocity of its bullets. Even though this is not physically correct, it is fine for our game.

required to work out a method to add an arbitrary series of bullets to a ship. For computing the total mass of a ship's collection of bullets, you can assume that all bullets have a radius of $3km$. Each time a ship fires a bullet, your implementation shall create a new bullet with a radius of $3km$, and position it as described above.

1.3 The Class Bullet

Each bullet is located at a certain position (x, y) . A bullet may be associated with a specific **Ship**, either being loaded on that ship or being fired by that ship. A bullet may be associated with a specific **World** under the constraints specified in Sec. 1.1. A bullet cannot be held by a world and a ship at the same time, nor can it be loaded on several ships at the same time. If a bullet is not loaded on a ship or associated with a world, it is positioned in an unbounded two-dimensional space. Both x and y are expressed in kilometres (km). All aspects related to the position of a bullet shall be worked out **defensively**.

Each bullet has velocities v_x and v_y (even if it is not associated with a world) that determine the bullet'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 bullet, 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 bullet, but it will always be less than or equal to c . All aspects related to velocity must be worked out in a **total** manner.

The shape of each bullet is a circle with finite radius σ (expressed in kilometres) centred on the bullet's position. The radius of a bullet must be larger than 1 and never changes during the program's execution. In the future, this lower bound may change, however it will always remain the same for each bullet and its value will be positive. All aspects related to the radius must be worked out **defensively**.

Each bullet has a mass expressed in kilograms (kg). For simplicity, we assume the mass density ρ of each bullet is the same, namely $7.8 \cdot 10^{12} kg/km^3$. The mass m of an bullet can hence be computed as follows:

$$m = \frac{4}{3}\pi r^3 \rho$$

where r is the radius of the bullet. You may assume that the mass of a bullet will always be much smaller than `Double.MAX_VALUE` in Java. All aspects related to the mass and mass density must be worked out **totally**.

The class **Bullet** shall provide a method **move** to change the position of the bullet based on its current position, its velocity and a given time duration Δt . The given duration Δt shall never be less than zero. If the

given duration is zero or the bullet's velocity is zero, the bullet shall keep its current position. As this method affects the position of the bullet, it must be worked out **defensively**.

The class **Bullet** shall provide methods to inspect the position, velocity, radius, density and mass. In addition, the class should provide a method that returns its source, if any, i.e. the ship that fired the bullet.

1.4 The Class **MinorPlanet**

Each minor planet is located at a certain position (x, y) . A minor planet may be associated with a specific **World** under the constraints specified in Sec. 1.1. If a minor planet is not associated with a world, it is positioned in an unbounded two-dimensional space. Both x and y are expressed in kilometres (km). All aspects related to the position of a minor planet shall be worked out **defensively**.

Each minor planet has velocities v_x and v_y that determine the minor planet'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 minor planet, computed as $\sqrt{v_x^2 + v_y^2}$, shall never exceed a maximum that is positive and never exceeds the speed of light c , $300000km/s$. A minor planet's maximum speed never changes during its lifetime. All aspects related to velocity must be worked out in a total manner.

The shape of each minor planet is a circle with finite radius σ (expressed in kilometres) centred on the minor planet's position. The radius of a minor planet must at all times be at least 5 km. That lower bound will not change in the future. The game distinguishes between two types of minor planets, known as asteroids and planetoids. The radius of an asteroid is fixed. However, planetoids shrink during travel time. More in particular, the radius of a planetoid is equal to its initial radius diminished with 0,0001% of the total distance travelled by that planetoid. If the radius of a planetoid would shrink below 5 km, the planetoid dissolves. All aspects related to the radius must be worked out **defensively**.

Each minor planet has a mass expressed in kilograms (kg). For simplicity, we assume the mass density ρ of asteroids to be $2.65 \cdot 10^{12}kg/km^3$, and of planetoids to be $0.917 \cdot 10^{12}kg/km^3$. The mass m of a minor planet can hence be computed as follows:

$$m = \frac{4}{3}\pi r^3 \rho$$

where r is the radius of the minor planet.

A minor planet can move. The class **MinorPlanet** shall provide a method **move** to change the position of the minor planet based on the current position, velocity and a given time duration Δt . The duration Δt shall never be less

than zero. If the duration is zero or the minor planet's velocity is zero, the minor planet shall keep its current position. As this method affects the position of the minor planet, it must be worked out **defensively**.

Students working alone: You must not work out planetoids as a kind of minor planet. You must only extend your project with asteroids. Ideally, it should be possible to extend your project with minor planets without having to change anything to the classes covering the world, ships and bullets. If this is not the case, we strongly advise you to restructure your code first, such that it indeed becomes possible to extend the game with minor planets.

Hint: Have a look at how new types of expressions can be added to the hierarchy of expressions, without a need to change existing classes [Exercise on calculator]. If changes to the specification are needed, you should also have a closer look at the substitution principle of Liskov.

1.5 Collisions

Ships, **minor planets** and bullets that are positioned in the same world may collide with each other and with the boundaries of the world they are in. Ships, **minor planets** and bullets that are not positioned in a world, do not collide. Your solution should include methods to predict and resolve collisions.

To account for rounding issues with the **double**-representation of real numbers in Java (cf. Sec. 2), your implementation shall employ a notion of *apparently collide* whenever checks for collisions of two objects are required: Two objects A and B *apparently collide* if the distance between the centres of A and B is between 99% and 101% of the sum of the objects' radii σ_A and σ_B .

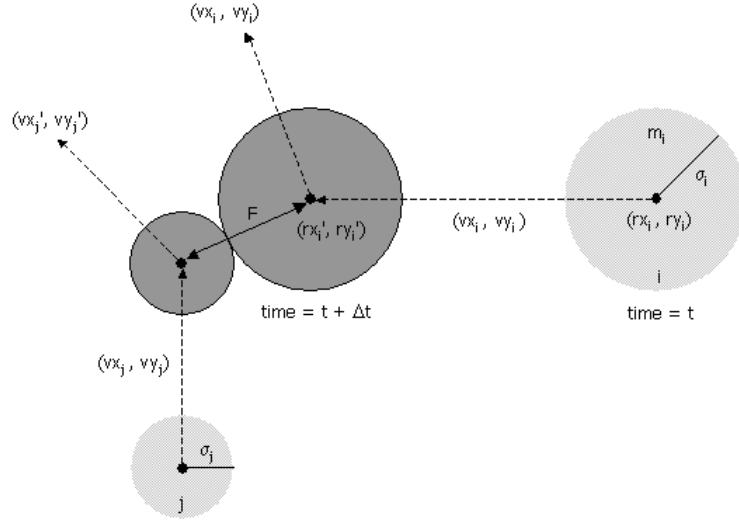
1.5.1 Collision Prediction

Entities (i.e., ships, **minor planets** and bullets) may collide with each other and the following section specifies the collision behaviour. You must add functionality to detect overlapping and to predict collisions. To do so, **the following methods must be applicable to ships, minor planets and bullets**:

- **getDistanceBetween** returns the distance in between two entities. The distance may be negative if both entities overlap. The distance between an entity and itself is zero.
- **overlap** returns **true** if and only if two entities overlap. An entity always overlaps with itself.

- `getTimeToCollision` shall return when (i.e. in how many seconds), if ever, two entities will collide. `getTimeToCollision` shall return `Double.POSITIVE_INFINITY` if the entities never collide. This method does not apply to entities that overlap.
- `getCollisionPosition` shall return where, if ever, two entities will collide. The method shall return `null` if the entities never collide. This method does not apply to entities 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 entities 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 entities i and j at the moment of contact, say $t + \Delta t$. When the entities 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 entities 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 Δt , selecting the physically relevant root, and simplifying, we obtain an expression for Δt 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.

Ships, [minor planets](#) and bullets collide not only with each other but also with the boundaries of the space they are in. Your solution should include one or more methods to determine when (i.e. in how many seconds) and where (i.e. which position at the boundaries of the world), a ship, [a minor planet](#) or bullet collides with a boundary (if ever), or with some other entity (if ever). All these methods must be worked out in **total** way, returning infinity (for time and positions), respectively `null` (for entities) if no collision will ever occur.

Predicted collisions between two entities or between an entity and the boundaries of the world are only correct if the movement of these entities is not influenced by other entities nor by collisions they may have with the boundaries of the world they are in. The methods to predict collisions will not take such influences into account. In other words, a predicted collision is only guaranteed to be correct if it is the first collision that will take place in the world, and if no other ships, [minor planets](#) or bullets are added to that world in the meanwhile.

1.5.2 Resolving Collisions

Resolve collisions as follows:

- When two ships collide, they bounce off each other. That is, the velocity of both ships is updated to reflect the collision.

- When two minor planets collide, they bounce off each other. That is, the velocity of both minor planets is updated to reflect the collision.
- When a bullet hits its own ship, the bullet is reloaded on the ship, with its position equal to the centre of the ship.
- When a bullet hits another ship, an asteroid, a planetoid or another bullet, both the bullet and the other entity die. This will also be the case if bullets hit other kinds of entities that may be added to the game later on.
- When a ship collides with an asteroid, the ship dies. The asteroid is not affected.
- When a ship collides with a planetoid, the ship is teleported to some random position in the world. If that position is occupied by some other entity, the ship dies. The planetoid is not affected.
- Ships, minor planets and bullets bounce off boundaries. More specifically, when an entity collides with a horizontal boundary (e.g. the top of the world), negate the y component of its velocity. For example, when a ship with velocity $(5, 8)$ collides with the top boundary, change its velocity to $(5, -8)$. When an entity collides with a vertical boundary, negate the x component of its velocity.
- When a bullet collides with a boundary for the third time, the bullet dies. That is, a bullet bounces off boundaries only twice. In the future, the maximum number of boundary bounces may vary from bullet to bullet.

When two ships or two minor planets collide, they bounce off each other. The new velocity of the ships, respectively of the minor planets i and j can be computed as follows:

$$(v_x^i, v_y^i) = (v_x^i + J_x/m_i, v_y^i + J_y/m_i)$$

$$(v_x^j, v_y^j) = (v_x^j - J_x/m_j, v_y^j - J_y/m_j)$$

where

$$J_x = \frac{J\Delta x}{\sigma}$$

$$J_y = \frac{J\Delta y}{\sigma}$$

$$J = \frac{2m_i m_j (\Delta v \cdot \Delta r)}{\sigma(m_i + m_j)}$$

Note that the mass of an object influences collisions resolution.

Ideally, the additional cases in resolving collisions should not have any impact on the existing classes **World**, **Ship** or **Bullet**. The additional rules should be handled in the classes covering minor planets.

Hint: Have a look at how new cards could be introduced in the game of thieves using double dispatch.

1.6 Death

Ships, **minor planets** and bullets can die when they collide with each other or with the boundaries of their world. When an entity dies, it is removed from its world (if any). In addition, when a planetoid dies, it spawns two asteroids, provided the planetoid is located within a world and provided the planetoid's radius is larger than or equal to 30 kilometres. The radius of each child is equal to half of the radius of the parent planetoid. The direction of the velocity of the first child is determined at random. The other child moves in the opposite direction. The speed of their velocities is 1.5 times the speed of their parent. Finally, both children are placed at a distance $r/2$ (where r is the radius of the parent) from the center of the parent planetoid. The centres of the parent and of both children should lie on a single line.

It shall further be possible to destroy **Worlds**. When a world is destroyed all its ships, **minor planets** and bullets are removed from it. The ships and bullets themselves are not destroyed. However, they no longer belong to a world. A world never contains dead ships, asteroids or bullets.

1.7 Programs

A program consists of an arbitrary number of function definitions followed by the program body. That body can be a simple statement such as an assignment statement. It can also be a structured statement such as a while statement or a block (a sequence of statements). As an example, consider the simple program shown below.

```
def incr {
  return $1 + $2;
}
d := 0.2;
while (1 < 2) {
  turn d;
  d := incr(d,0.1);
  fire;
}
```

This program consists of the definition of a function **incr** followed by a block that constitutes the body of the program. The function involves two

parameters and a return statement as its body. The `return` establishes the value that the function will return. The body of the program is a block that consists of two assignment statements followed by a while statement. The body of the while statement is again a block consisting of three statements: a turn statement, an assignment statement and a fire statement.

Programs may introduce variables to store the results of computations. The program above introduces the global variables `d` and `b`. Global variables can be declared at any point in the program. However, they must first be assigned before being used in an expression. Variables are not explicitly assigned a type but take the type of the first assignment. All further assignments to the same variable must be of the same type.

No documentation (formal nor informal) nor tests are required for all the classes related to this part of the assignment.

1.7.1 Program Execution

A ship can store a program that determines its actions. The program associated with a ship executes as time advances. In other words, if a ship has to be advanced over some time `T1`, its program will run for that same period of time. Each execution of an action statement (turning, enabling the thruster, disabling the thruster, firing a bullet or doing nothing) takes 0.2 seconds. Execution of all other kinds of statements does not take any time. Whenever a program has less than 0.2 seconds to execute, it is put on hold. If the program must be executed for some additional amount of time `T2`, it continues at the point it was put on hold. The time left from the previous execution is added to the new amount of time `T2`.

As an example, consider the program shown above, and assume it must be executed for 0.5 seconds. The program will initialise the variable `d` to 0.2. It then starts executing the while statement. It will instruct its ship to turn over 0.2 radians, then increment the variable `d` by 0.1, and subsequently instruct its ship to fire a bullet. At that point, the program only has 0.1 seconds left to execute, and it is put on hold. If the program is subsequently executed for another 0.65 seconds, it will instruct its ship to turn over 0.3 radians, increment the variable `d` with 0.1, instruct its ship to fire a bullet, followed by another turn over 0.4 radians. At that time, it only has 0.15 seconds left to execute (0.1 from the previous execution and 0.05 from the current execution), meaning that the program is put on hold again.

Hint: Have a close look at how expressions got evaluated and transformed into postfix notation in the calculator exercise. A similar structure can be used to execute programs loaded on ships.

1.7.2 Statements

The syntax of a statement `s` in Backus Normal form (BNF) notation is as follows:

```
s ::=
    x := e;
  | while(e) { s }
  | break;
  | return e;
  | if (e) then { s } else { s }
  | print e;
  | s*
  | action

action ::=
    turn e;
  | fire;
  | thrust_on;
  | thrust_off;
  | skip;
```

That is, a statement is either an assignment statement, a while statement, a break statement, a return statement, an if-then-else statement in which the else-part is optional, a print statement, a block (a sequence of zero or more statements) or an action statement. There are five different kinds of action statements: turning, shooting, enabling the thruster, disabling the thruster and doing nothing (skip).

The statement `print e` shall output the result of evaluating the expression `e`. The program will print the textual representation of that value (resulting from Java's `toString` method) on Java's standard output stream. When the execution of a program terminates, the program must also return a list of all the values (not their textual representation) it has printed out.

Break statements may only occur in the body of while statements. They terminate the execution of the immediately enclosing while statement. Return statements may only appear in the body of functions. They immediately stop the execution of the function at stake, and return the value specified as part of the return statement.

Students working alone: You do not have to support break statements nor action statements related to the thruster of a ship.

1.7.3 Function Definitions

The syntax of a function definition in BNF notation is as follows:

```
function_definition ::=
    def f
    {s}
```


A function definition starts with the keyword `def` followed by the name of the function and terminated with the body of the function. Functions cannot have the same name as global variables. If a program contains several definitions of a function with the same name, the lexicographically last definition applies. The body of the function is a block enclosed in curly brackets. The execution of the body of a function must always terminate with a return statement. In other words, the body of a function is not implicitly extended with a return statement that returns some default value. The body of a function cannot contain any action statements. As a consequence, the execution of a function will never be timed out. The body of a function can also not include any print statements.

Notice that the definition of a function does not include a parameter list, as is common in most programming languages. Each function has an arbitrary number of parameters, denoted `$1`, `$2`, etc. The parameters of a function can only be read during the execution of its body. Parameters cannot occur at the left-hand side of assignments. Each time a function is invoked, a series of actual arguments is supplied. The first actual argument is bound to the parameter `$1`, the second actual argument is bound to the parameter `$2`, etc. A function may be invoked with more actual arguments than the function actually needs. However, if during the execution of the body of a function a value for one its parameters is missing, a runtime error is to be raised.

Functions are allowed to introduce local variables. In fact, each assignment in the body of a function is an assignment to a local variable. The body of a function only has read access to the variables defined at global scope. The body of a function can also invoke other functions defined at global scope. However, if some local variable of some function `f` has the same name as some global variable or some function, that global variable or that function is out of reach for the function `f`. Functions may directly or indirectly invoke themselves.

You must only work out function definitions and function invocations if you want to get a score of 16 or more for the final project.

1.7.4 Expressions

The syntax of expressions `e` in BNF notation is as follows:

```
e ::=
  x
| c
| entity
| (e)
| - e
| e + e
```

```

| e * e
| f([e(,e)*])
| ! e
| sqrt e
| getx e
| gety e
| getvx e
| getvy e
| getradius e
| getdir
| e < e
| e == e

```

```

entity ::=
    null
| self
| ship
| asteroid
| planetoid
| bullet
| planet
| any

```

An expression is either a variable **x**, a formal parameter **p**, a double constant **c**, an entity, an expression enclosed in parenthesis, a change of sign, an addition, a multiplication, an invocation of a function **f** involving a list of actual arguments, a logical negation, a **sqrt**, a **getx**, a **gety**, a **getvx**, a **getvy**, a **getradius**, a **getdir** or a comparison (less than or equal to). The arithmetic operators and the less than operator are only defined for values of type double. The logical negation is only defined for boolean values. The equality operator is defined for all possible kinds of objects. It yields false if both objects are not of the same type.

The expression **sqrt** returns the square root of its argument. The expressions **getx e**, **gety e**, **getvx e**, **getvy e**, **getradius e** respectively compute the x-coordinate, y-coordinate, velocity along the X-axis, velocity along the Y-axis and radius of the entity **e**. The expression **getdir** returns the direction of the ship executing the program.

An entity is either **null** (no entity), **self** (the ship that executes the program), **bullet** (one of the bullets fired by the ship that executes the program), **ship**, **asteroid**, **planetoid**, **planet** (respectively the ship, the asteroid, the planetoid or the minor planet closest to the ship executing the program), or **any** (an arbitrary entity in the world of the ship that executes the program). If no entity matches an entity expression at the time of evaluation, the expression yields **null**. Functional aspects of Java 8 shall be used as much as possible to implement entity expressions. If that turns out to be impossible, lambda expressions and higher-order functions must be used in

at least one other part of your project.

Students working alone: You may choose not to support expressions in parenthesis, multiplications, logical negations, entity expressions `ship`, `asteroid`, `planetoid` and `planet`, nor expressions involving `sqrt`, `getvx`, `getvy` and `getdir`.

1.7.5 Type Safety and Well-Formedness

Programs defined in the above syntax employ three different types of expressions, functions and variables: `double`, `boolean` and `entity`. Variables of type `double` or of type `boolean` adopt value semantics. Variables of type `entity` adopt reference semantics. The above syntax does not narrowly define when each type may be used. During the execution of a program, you will check that only valid types are used at times expressions are evaluated. As an example, an addition involving an entity at its left-hand side and a double at its right-hand side is illegal. As another example, a while statement involving an expression of type `double` as its controlling expression is illegal. Execution of a program immediately stops as soon as such an error is encountered. This will also be the case if a break statement is executed that is not included in a while statement, if a return statement is executed that is not included in the body of a function, if the execution of the body of a function terminates without returning a value.

If you want to score 17 or more on this project, you will try to prevent the creation of faulty programs as much as possible. You must use generic classes and/or interfaces for that purpose. The Java compiler must then complain if we try to build a program that has typing errors. We only expect those messages if we create programs directly by means of constructors in your hierarchy. Note that we do not expect you to prohibit Java programmers to build incorrect programs by using the program factory, or by means of reflection or by means of raw types underlying generic classes. For that reason, you cannot leave out any of the runtime tests described above. In other words, you may not assume that all programs to be executed have been created by means of constructors that you supply as part of your classes.

Examples of trivial typing errors are `4.0 + self`, `5.0 < self`, or `4.0` as condition of a `while`. An example of an error that we do not expect you to catch in this way is `a := self; if a < 0.0 print 5.0`. This part of the assignment serves to convince us that you can define generic classes. Its purpose is not to work out an in-depth static analysis of programs.

1.7.6 Parsing

The assignment comes with a number of example programs stored in text files. Reading a text file containing a program and converting it from its textual representation into a number of objects that represent the program in-memory is called *parsing*.

The assignment includes a parser. To parse a `String` object, instantiate the class `ProgramParser` and call its `parse` method. This method constructs an in-memory representation of the program by calling methods in the `IProgramFactory` interface. You should provide a class that implements this interface. The parser was generated using the ANTLR parser generator based on the file `AsteroidsParser.g4`. It not necessary to understand or modify this file.

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 number³

0.200000000000000011102230246251565404236316680908203125

When performing complex computations in type **double**, rounding errors can accumulate and become arbitrarily large. The art and science of analysing

³You can check this by running `System.out.println(new BigDecimal(1.0/5.0))`. Rounding also affects literals, e.g., a new `BigDecimal(0.2)` will not equal the literal `0.2`.

computations in floating-point types (such as **double**) to determine bounds on the resulting error is studied in the scientific field of *numerical analysis*.

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

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

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

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

3 Testing







Write a JUnit test suite for all the non-trivial public methods of your project as far as they are related to the world and the entities in it. We do not expect tests for simple methods such as setters for properties ascribed to the world or its entities. You may also leave out tests for other methods that are trivial in your opinion. Include your test suite in your submission. As mentioned before, no tests are required related to programs and their execution.

Included in the given files, you will also find an elaborate set of tests that covers the entire project. This collection not only serves to check the correctness of your code but also to clarify details of the game in general, and of the programming language in particular. Any questions or remarks that you have on this test suite must be directed to Prof. Steegmans directly (as it is part of the overall evaluation of the project). The collection may change and grow until the 15th of May.

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. Similar to the GUI for part 2, you should import this JAR file as an archive into the root of your existing project from part 2. This will only overwrite resources in folder `src-provided`. That folder 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, update your class `Facade` in package `asteroids.facade` to implement the provided interface `IFacade` from package `asteroids.part3.facade`. This interface extends the `IFacade` interface from part 2 with additional methods (and deprecates some others). `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 `Part3` in package `asteroids.part3`. You now have the choice between “Player vs Asteroids” and “Player vs AI”, the latter involves loading and executing a program as described in Sec. 1.7. Check `Part3.java` and `Asteroids-Frame3.java` to see how a program is loaded and the parser is invoked. By default, `resources/programs/program.txt` will be loaded. After starting the game, you can press keys to modify the state of the program. The command keys are  and  to turn,  to enable/disable the thruster,  to fire,  to show collision points, and  to terminate the program.

You can freely modify the GUI as you see fit. However, the main focus of this assignment are the classes described in this assignment. 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 **25th of May 2017 at 11:59 PM**. You can generate a JAR file on the command line or using Eclipse (via `export`). The name of your JAR must include your first and last name, as well as the first name and

last name of your partner, if any (e.g. `EricSteegmans_KoenYskout.JAR`). 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, if any, in the comments of your solution. When submitting via Toledo, make sure to press OK until your solution is submitted!