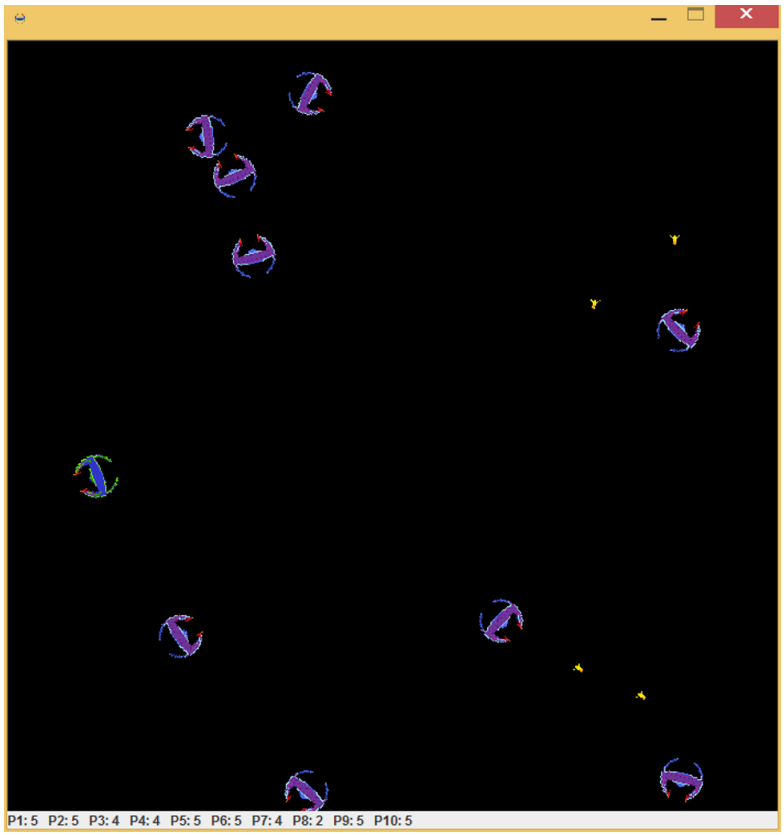
Object Oriented Programming - Exercise 2: Space Wars

1 Objectives

By the end of this exercise, you'll have a fully working space wars game. Much of the code is supplied by us - your own mission is to implement some different kinds of spaceships.

2 Game Description



The screenshot above shows a few spaceships fighting for survival. At the bottom of the screen the number of deaths of each spaceship is displayed. Each game should include at most one human controlled spaceship (the different colored one), and any number of computer controlled ones.

There are different kinds of spaceship behaviors. Let us denote them by letters:

- h - Human controlled ship (controlled by the user).
- r - Runner: a ship that tries to avoid all other ships.
- b - Basher: a ship that deliberately tries to collide with other ships.

- a - Aggressive: a ship that tries to pursue other ships and fire at them.
- d - Drunkard: a ship with a drunken pilot. Explain in your README how you implemented this behavior.
- s - Special: a ship with some interesting behavior that you will define yourselves and explain in your README.

Detailed specification of these behaviors will follow. Your job is to implement all of the ship types listed above (you may also add additional files required for your implementation, but additional ship types will not be checked).

The game is launched from the command line as follows:

`java SpaceWars <space_ship_type> <space_ship_type> [<space_ship_type> ...]`

where the spaceship types are one letter arguments as denoted above. For example:

```
java SpaceWars h a d
```

opens a new game window with one human controlled ship, one aggressive ship, and one drunkard. The notation above means that at least two types are required in order to run the game.

You can either supply these arguments by running the binaries IDEA compiled (which are in your workspace), or by adding them in IDEA: `Run->Edit configurations->Program arguments`.

3 Provided Code

You are provided with two kinds of code: a "black-box" helper package named `oop.ex2` which you will "import" in your code, and some code for the actual game - most of which is only partially implemented. You can download all of it from Moodle.

3.1 The helper package `oop.ex2`

Within this package are numerous java files - however only a few of the classes are part of its API (only public classes are visible from outside the package). Moreover, some of the public classes provide API which is already utilized by code given to you. This leaves only two classes in the package which should concern you. In order to use them, you should go to `File->Project Structure->Libraries`, click the plus sign, select `Java` and browse for the supplied `jar`. For each class that uses the package add at the beginning of the file (before the class's definition):

```
import oop.ex2.*;
```

- `SpaceShipPhysics` - Represents the physical state of a spaceship (its position, direction, speed). Every spaceship needs to have exactly one instance of this class ("has a" relationship). When created (through the default constructor), the SpaceShipPhysics object is initialized randomly. A new object of this class should be created in the following cases:
  a) When a spaceship is created
  b) When a spaceship dies and restarts
  c) When a spaceship teleports

All the updates of the spaceship's position are done via this object. See the API of this class.

- `GameGUI` - This class is responsible for graphical output and user input.
  - It defines four images: a human controlled ship, a computer controlled ship, human controlled ship with shields activated and a computer controlled ship with shields activated. As later described, each spaceship will have a `getImage()` method that returns its representing image - this image should be one of these four (by the way, to implement the `getImage()` method of a spaceship you'll have to `import java.awt.Image;`).
  - In addition, this class has methods that detect user input - these should be used by your program to control the human-controlled spaceship. See the full API of this class.

3.2 Additional supplied code

- `SpaceWars` - The 'driver' of the game - it runs and manages it with its main method (note that in the game launching instructions above, this is the class we actually ran). You will not include this class in your submission (this is the only file you shouldn't submit), and thus you **CANNOT CHANGE IT**. All your code should compile with this driver as is. See the API of this class.
- `SpaceShip` - Currently only lists the minimal API that each ship in the game should expose. You may expand this API (keeping in mind the minimal API principle), but you cannot remove any methods from it. You may also make this class abstract or an interface, as long as `SpaceShip` is a common **type** for all ships. Additionally, you may also decide to make any of the methods in this API `abstract`, but may not change their signature in any other way. See the API of this class.
- `SpaceShipFactory` - This class has a single static method (`createSpaceships(String[])`), which is currently empty. It is used by the supplied driver to create all the spaceship objects according to the command line arguments. You will implement this static method. See the API of this class. E.g., if the supplied array contains the strings "a" and "b", the method will return an array containing an Aggressive ship and a Basher ship.

> A factory is a standard name for an entity which is responsible for creating instances. You will learn more about different kinds of factories later in the course.

4 The rules of the game

All the spaceships in the game should have the following attributes:

- A `SpaceShipPhysics` object (from the helper package), that represents the position, direction and velocity of the ship.
- A maximal energy level.
- A current energy level, which is between 0 and the maximal energy level.
- A Health level between 0 and 25.

4.1 Health

- Begins at 25.
- Being shot at reduces health by 1.
- Collisions between two spaceships reduces both ships' health by 1.
- However, if a ship has its shields up - it will take no damage (from neither shots nor collisions).
- A ship is considered dead when its health reaches zero.

4.2 Energy

- The maximal energy level starts at 250, while the current energy level begins at 200.
- "Bashing" is when the ship has its shields up and collides with another ship. When a ship bashes another, its maximal energy level goes up by 20, but the current energy level is reduces by 15 (for example if the ship's energy is 24 out of 190, colliding with another ship while the shields are up brings the energy to 9 out of 210).
- Getting hit (either getting shot at or colliding) while the shields are down or while the shields are up but there is not enough energy for the cost, reduces the maximal energy level by 5 and the current energy by 10.
- The current energy level is constantly charging: it goes up by $\lfloor 2 \cdot \frac{CurrentVelocity}{MaximalVelocity} \rfloor + 1$ every round (rounds will be explained in a bit), up to the ship's current maximal energy level. In other words, the charge is +1 when velocity below half the maximal velocity, +2 when above half and +3 when at maximal velocity.
- Getting shot while the shields are up reduces the current energy by 2.
- Firing a shot costs 15 energy units.
- Teleporting costs 100.
- If the ship's shields are up in a certain round, they consume 3 energy units for that round.
- Energy levels are non-negative. Energy reduction from events that require more than the available energy will not take place. If the current energy level is above the maximum, it changes to the maximum.

4.3 Spaceships Actions

The game proceeds in rounds (multiple rounds take place per second). In each round, each of the spaceships may perform any combination of the following actions (or none of them):

- Accelerate: the spaceship will accelerate a bit in the direction it's facing.

- Turn: the spaceship will turn slightly to the left or right.

  Note: both the turning and acceleration actions should be performed through the `SpaceShipPhysics` *object, using a single call to its* `move()` *method. This method should be called exactly once per round even if the spaceship does not accelerate or turn, or if it performs both.*

  Note 2: If both the right and the left button were pressed, the ship will not turn at all.

- Teleport: The spaceship will disappear and reappear at a random location. This should be done by creating a new `SpaceShipPhysics` object (which is initialized randomly) for this ship.

- Fire a single shot: This can be done by calling the `addShot()` method of the `SpaceWars` driver. After firing, the ship's guns cannot be used for a period of 7 rounds; e.g. if a ship has fired on the first round of the game, it can shoot again on the eighth round.

- Turn on its shield (for the current round only!).

Since several actions can take place in a single round for the same ship, we define their order of precedence:

1. Teleport
2. Accelerate and turn (happen at the same time)
3. Shield activation
4. Firing a shot
5. Regeneration of energy of this round

This means that if both the teleport button and some of the movement buttons were pressed on the same round, the ship will first teleport and then will update its location (in relation to the teleport point) according to the movement parameters for that round.

Changes to the ship's health and maximal energy levels are applied when the object is notified of their respective events by the supplied driver class.

4.4 Death

When a ship dies the `SpaceWars` driver will call the ship's `reset()` method. The spaceship should then reappear in a new random position, with its initial maximal health, current health, maximal energy and current energy values, as if it was only now created. Additionally, the firing cool-down resets; meaning that if - on time of its death - the ship could not fire because fewer than 7 rounds have passed since its last shot, any counter tracking this information is reset, and the ship can immediately fire on the round of its rebirth.

The `SpaceWars` object will automatically keep track of the number of times each ship was destroyed. The game goes on indefinitely, until the escape key is pressed or the window is closed (this functionality is already built in).

5 The different types of spaceships you should implement

1. **Human controlled**: Controlled by the user. The keys are: left-arrow and right-arrow to turn, up-arrow to accelerate, 'd' to fire a shot, 's' to turn on the shield, 'a' to teleport. You can assume there will be at most one human controlled ship in a game, but you're not required to enforce this.

2. **Runner**: This spaceship attempts to run away from the fight. It will always accelerate, and will constantly turn away from the closest ship. If the nearest ship is closer than 0.25 units, and if its angle to the Runner is less than 0.23 radians, the Runner feels threatened and will attempt to teleport.

3. **Basher**: This ship attempts to collide with other ships. It will always accelerate, and will constantly turn towards the closest ship. If it gets within a distance of 0.19 units (inclusive) from another ship, it will attempt to turn on its shields.

4. **Aggressive**: This ship pursues other ships and tries to fire at them. It will always accelerate, and turn towards the nearest ship. When its angle to the nearest ship is less than 0.21 radians, it will try to fire.

5. **Drunkard**: Its pilot had a tad too much to drink. We leave it to your creativity to define the ship's exact behavior, but it must include randomness and should definitely be amusing to fight against.

6. **Special**: Come up with a unique behavior which is interesting and/or successful and/or makes the game more fun.

Note: Methods to get the distance or angle from another spaceship are part of the `SpaceShipPhysics` class. The closest spaceship can be found using the `getClosestShipTo()` method of the `SpaceWars` game driver.

6 Design

The whole design of this exercise is completely up to you. You're required to explain your considerations and choices in your README, and a noticeable chunk of your grade will reflect your design. In "real life" a good design is priceless, and its effect on development time, ease of maintenance and the amount of bugs could not be overstated. It's important to us that you put some thought into yours: a design into which little thought was put would lose more points than an imperfect design in which the student considered a few choices and didn't make the best one due to lack of experience.

Possible design principles that you can use in your design are inheritance, abstract classes, interfaces and/or composition (the "has a" relation). Whatever your design is: keep it simple and keep your API minimal. A good design is simple and intuitive, yet keeps the code short, unrepetitive, and easy to make changes to.

7 FAQ

- When the shield button is pressed, the shield is on for the current round. After that round, if the button is not pressed, the shield should be removed.
- You don't have to check the validity of function input unless noted otherwise.
- You may import the Java `Math` library and use any and all of its methods.
- You can use the `protected` modifier when expanding the API, but you can not change the given modifiers to `protected`.
- You **cannot** use reflection. Generally, it is almost always the wrong design decision.
- You are allowed to use Java's `Random`.
- You are allowed to add your own image files to your `jar` files. Keep them small, though.
- You're not allowed to change the `SpaceWars` file. And since you can't submit it, any such changes will not be reflected in our testing anyway.

8 Grading

This exercise will have no automatic tests. The graders will go over your code, your README and your design, and play your game.

9 Javadoc

Document your code using the Javadoc documentation style, as described in the coding style guidelines and exampled in the code. You should check yourself by invoking: `javadoc -private -d doc *.java` within your project directory or by running IDEA's `Tools -> Generate JavaDoc...` in the `Project` menu (but the documentation files shouldn't be submitted). Either method should succeed without warnings or errors. The generated `html` files can be viewed using a browser.

10 README

- Explain your design. What were your considerations? What are your design's advantages and disadvantages in terms of:
  - Extensibility - The ability to add new functionality or features.
  - Modularity - The degree to which the program divides to distinct components which interact via their AP.
  - Intuitiveness and understandability - Note that a program's understandability does **not** refer to the code's readability, but rather to the ease of understanding how the program is built and designed.

  - Representation of the classes and their relationships - Both the choice of what concepts and constructs are represented by classes and the relationships between these classes make sense.

- Describe the behavior of the drunkard ship and special ship.
- Include any other comments regarding your implementation.

11 School Solution

You can try out the school solution by typing (only on CS lab computers):

`cd ~oop/bin/ex2 && SpaceWars <two_or_more_spaceship_types>`

12 Submission and Further Guidelines

Your `ex2.jar` should include (only) the following files:

- `SpaceShip.java`
- `SpaceShipFactory.java`
- All other java files you wrote (excluding `SpaceWars.java` and the supplied package)
- README

When executing the command
`javac -cp oop.ex2.jar *.java`
with your files in the current directory and the helper package `oop.ex2.jar` in the same folder, your files should compile with our driver and helper package without any warnings or errors.
Then, your program should run with the command
`java -cp .:oop.ex2.jar SpaceWars a h d` {*The specific arguments are just an example*}
On Windows system replace `:` with `;`, so run:
`java -cp .;oop.ex2.jar SpaceWars a h d`

13 Work Suggestions

- Begin today.
- You'll save LOTS of time and frustration by writing your documentation (Javadoc and other) AS YOU GO, instead of adding it just before submission.
- The same goes for any hand-made tests. Writing them after writing most of your code, or not at all, will not save you any time; it will only cost you time in debugging.
- Work carefully and patiently, without trying to have as much code written as quickly and dirtily as possible.

- After deciding on your design, implement it only to the extent of a runnable, very minimal game, which supports only one ship type, whose behavior is not even implemented yet.
- Finish implementing your design infrastructure and that ship-type's behavior so that the game works flawlessly for it.
- During this process:
  - Run the game as frequently as possible to check each added feature.
  - Only when the ship-type is fully implemented, documented, tested, and functional, proceed to the next type.
- Following these guidelines requires a lot of patience, but will save you lots of time debugging and writing documentation. Plus, once you're done with the last type you'll have the satisfaction of knowing that your code is documented, tested and functional.

Good Luck !