FLEET ADMIRAL’S HANDBOOK

UNIVERSAL SPACE FLEET COMMUNICATION PROTOCOL 1.0

PREFACE

Greetings,

Congratulations on your new appointment as fleet admiral. We entrust in you supreme control over your fleet so you can deliver nothing short of total victory.

This document hopes to tell you all you need to know about space warfare. It will describe to you the standard operating procedures and the protocol used to communicate with your spaceships in order to get you to the action as soon as possible.

Good luck.

Best wishes,

Fazli Sapuan

Grand Admiral of the Fleet

TABLE OF CONTENTS

Preface 2
General Overview 1
The Universe 3
Ship Production 4
Space Combat 5
Round Transaction Order 6
Victory Conditions 7
Tournament Set-up 8
Universal Space Fleet Communication Protocol 9
    Universe Object 9
    Planet Object 10
    Fleet Object 10

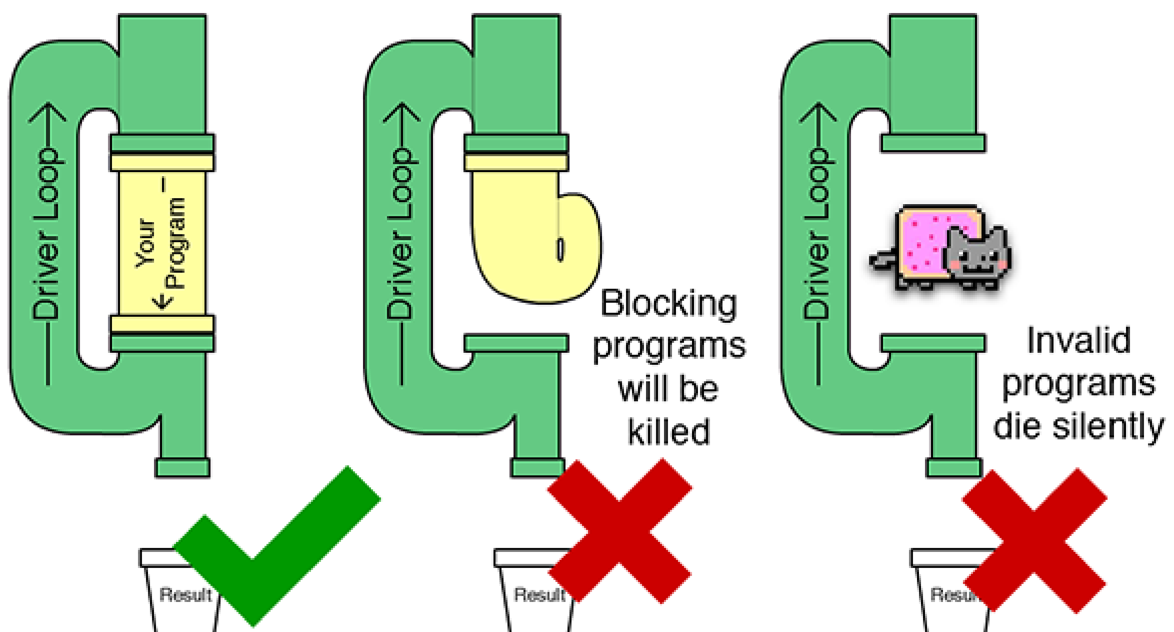
The game is a turn-based artificial intelligence game.

Each player must write a **JavaScript program** that will instruct the fleet of unmanned spacecraft to perform certain actions in a particular round.

At the start of a round, each player's program is executed with the state of the game called the *state of the universe*. Based on the information given in the state, each program will **commit a set of instructions to be performed**. There is no limit to the number of instructions that can be committed so long as the instructions are valid and are physically possible. The program, however, must return within a fixed **time constraint of 1 seconds** or the program's turn will be **skipped for that round regardless of any committed instructions.**

At the end of each round, the state of the *program* is preserved but the *state of the universe* will be overwritten by a new one.

The following diagram illustrates the expected control flow of your program:



You may structure your program in any way you like. This is the suggested design pattern for your program:

```
if (typeof init === "undefined") {
    // state initialisation goes here
    // ...
    // commit initial state instructions here

    var init = true;
}

// perform calculations here
// ...
// commit instructions here
```

Your program must be free from error. If your program triggers an error during its runtime, the execution of the rest of the program is not guaranteed.

The universe is the collection of all planets that exists in the game. In order to simplify space warfare, we make a couple of assumptions:

1. All planets lie on a flat 2D plane
2. All planets are stationary

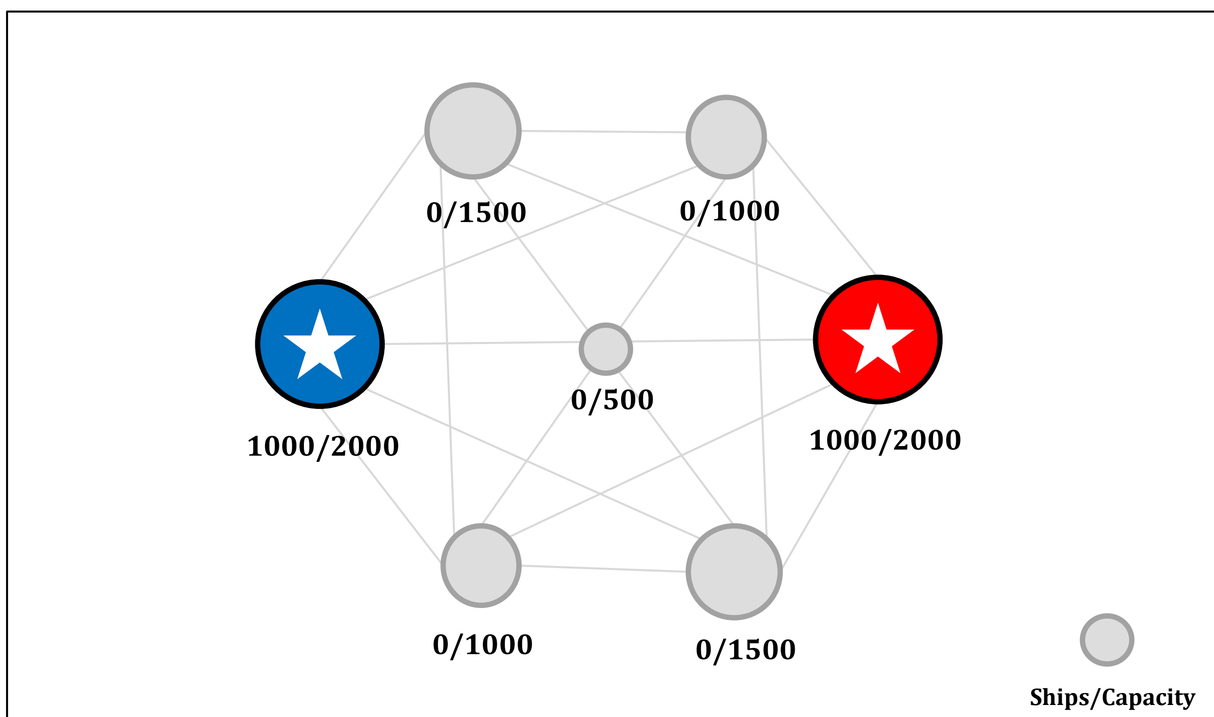
There are **at most 100 planets** in the universe and planets have different sizes which determine the physical capacity of a planet. Universes are **always rotationally symmetrical** and **always contain an odd number of planets**.

In order to aid navigation, the positions of the planets are described by the *x-coordinate* and *y-coordinate* of the planet relative to a fixed origin called the *centre of the universe*.

These coordinates are in a unit called the *astronomical unit*, *u*. All planets are **at most 100u away** from the *centre of the universe* effectively making the universe disc-shaped.

You start your war on a special planet called the *home planet*. Home planets are the largest planets available in the universe and are visually marked with a *capitol star*.

Example universe:



SHIP PRODUCTION

All captured planets **automatically** produce ships at the end of a round.

Players **have the option** to trade a flat fee of a fixed number of ships to upgrade a planet. Upgraded planets will produce spaceships at a higher rate than usual. The number of home planets held by the player will also affect the rate of production of all planets under the player's control.

The rate of production is defined as:

$$\frac{dy}{dr} = \left\lfloor 10 \times L \times \frac{9+h}{10} \right\rfloor, where:$$

$$y = spaceships\ on\ the\ planet$$
$$r = rounds\ elapsed$$
$$L = level\ of\ the\ planet$$
$$h = home\ planets\ held$$

The cost of upgrading a planet is as follows:

Level	Cost
2	0.3 x Capacity of planet
3	0.6 x Capacity of planet

Any excess ships above the capacity of the planet are destroyed due to *overpopulation*.

At the start of the match, each player is given a fleet of **1000 spaceships** at their respective home planets.

Spaceships **can be instructed** to travel from one planet to another.

When assigned a destination, spaceships **automatically** travel towards the destination planet at a **constant speed of 10u/round**. Due to technical limitations in spacefaring technology, spaceships can only travel in *straight lines* and cannot receive instructions while in *open space*.

Space combat doctrine requires spaceships to **travel in a group no smaller than 20**. The ships on-board computers will refuse to launch if this condition is not met.

Spaceships **automatically** capture planets as long as they exist on the planet at the end of the round. If a planet with an upgrade is captured, the planet is **downgraded by 1 level**.

Spaceships **automatically** engage enemy spaceships when on planets with a 1:1 mortality ratio.

ROUND TRANSACTION ORDER

As with all transaction-based systems, the order of which instructions are carried out in a transaction matters. For example, banks can carry out withdrawal instructions first before carrying out deposit instructions in order to charge overdraft fees.

Extrapolating that idea, when any ambiguities of order of the player's instructions and the steps in game state occur, the correct order is designed in such a way that the order that produces the worst possible outcome.

The order of a round's transaction is as follows:

1. All player commands
2. Automatic ship movements / planet captures with the following criteria in descending order of importance:
    a. The closer ships goes first
    b. The largest force goes first
    c. The ships that travelled the furthest goes first
    d. Coin flip (highly unlikely)
3. Automatic production of ships

VICTORY CONDITIONS

The game ends when **all** these necessary conditions are satisfied:

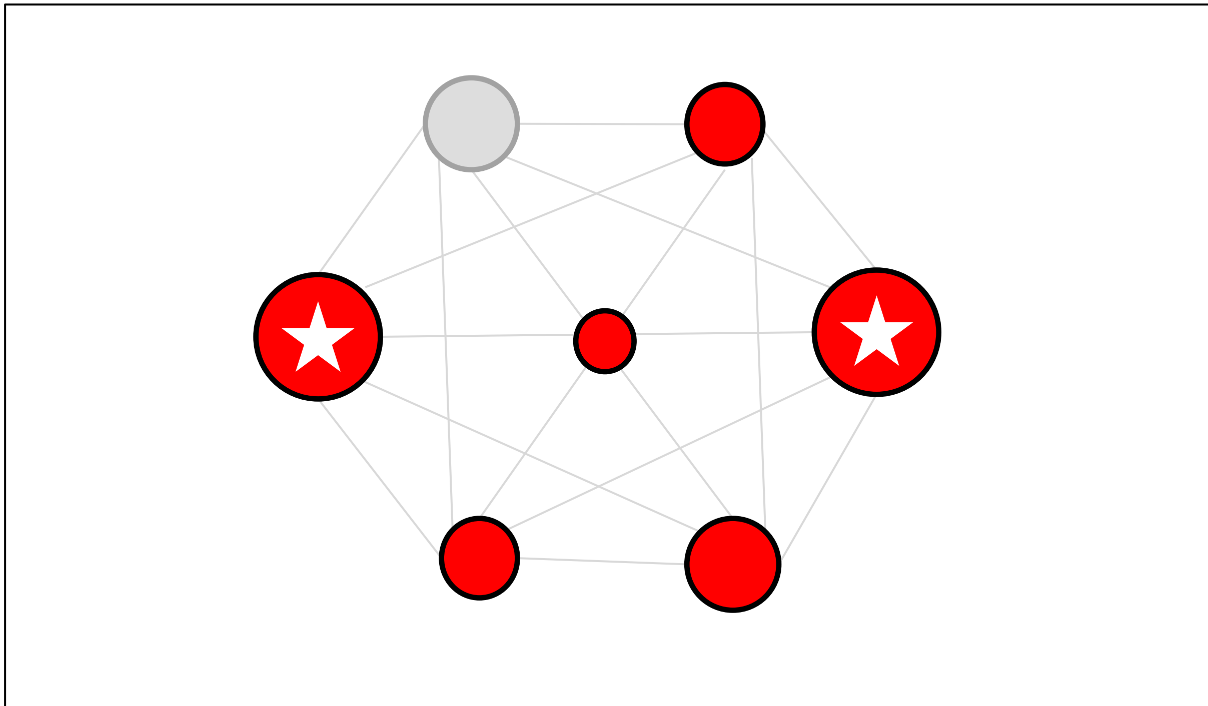
1. Only 1 player has the ability to produce ships in the next turn
2. Not more than 1 player have spaceships in the universe

In such an event, the *sole survivor* is deemed the winner of the match.

However, if these conditions cannot be met after 500 rounds, a *time-out* occurs. In the event of a *time-out*, the winner will be decided by the following tie-breakers in descending order:

1. Number of planets controlled by player
2. Size of space fleet
3. Rate of production of spaceships
4. Coin flip (highly unlikely)

Example Red victory:



Notice how red does not need to capture every planet in the universe.

TOURNAMENT SET-UP

In the *tournament*, participating programs are pit against each other 1-on-1 in a random match up.

Each participant starts with an Elo rating, *R* of 1000. After every match, a new Elo rating, R' is calculated using the following formula:

$$R' = R + K(S - E)$$

$$E = \sum_{i=0}^{N} \frac{1}{1 + 10^{\frac{R_i - R}{400}}}$$

$$where,$$

$$K = 10$$

$$S = number\ of\ wins$$

$$R_i = opponent's\ rating$$

$$N = number\ of\ opponents$$

Defeating an opponent with a higher rating will give a proportionally larger rating increase. Similarly, being defeated by an opponent with a lower rating will give a proportionally larger rating decrease. Notice how the combined ratings of both players do not change throughout the tournament.

The Elo rating is finalized approximately when enough matches are carried out. The winner of the tournament is the player with the highest Elo rating.

This is the API you need to use to communicate with the space fleet and radar systems.

You may also use the underscore.js supporting library in your program. Underscore.js provides functional programming functionality for your program, enabling you to write concise and elegant code using techniques like higher order functions. To see a complete list of supported functions, visit http://underscorejs.org/.

If your program encounters an error or times out, a message will be logged in the JavaScript console of your browser. You may also log messages to the console by using console.log(). These messages will be visible to anyone watching the match.

IMPORTANT: Only use these methods. Naked access is not illegal but unreliable and strongly discouraged.

Due to the freeze and thaw methods used by the server, the following keywords are **strictly reserved. DO NOT USE THESE KEYWORDS IN YOUR PROGRAM:**

- __IS_UNIVERSE__
- __IS_ROOT__
- "_UNIVERSE_" (n.b. this is a String)

The presence of these keywords can possibly (but not always) cause the interpretation of your program to fail, making you lose your turn. This is mainly because of corruption of your program's state, resulting in loss of data.

REMINDER: If you capture results from methods that return Arrays into your program's state, the array may contain references to objects that do not exist and are no longer valid. It is the responsibility of your program to detect these invalid occurrences.

UNIVERSE OBJECT

```
universe.getPlayerId()
```

Returns the integer *player_id* of the player.

```
universe.getNumOfPlayers()
```

Returns the number of players in the match.

```
universe.getPlanets()
```

Returns the array of planet objects representing all planets in the universe.

```
universe.getFleets()
```

Returns the array of fleet objects representing all groups of ships in the universe.

```
universe.actions.send(<ships>, <from>, <to>)
```

Commits a send action of integer *ships* ships, from planet object *from* to planet object *to*. The function returns boolean *true* if successful and *false* on error (e.g. not physically possible/not owner of planet).

```
universe.actions.upgrade(<planet>)
```

Commits an upgrade at planet object *planet*. The function returns boolean *true* if successful and *false* on error (e.g. insufficient ships/not owner of planet).

```
universe.actions.clear()
```

Clear all actions that had been commited up to this point.

PLANET OBJECT

```
planet.getX()
```

Returns the x-coordinate (in astronomical units) of the planet.

```
planet.getY()
```

Returns the y-coordinate (in astronomical units) of the planet.

```
planet.getSize()
```

Returns the capacity of the planet.

```
planet.getOwner()
```

Returns an integer *player_id*.

```
planet.getShips()
```

Returns the number of ships on the planet.

```
planet.getLevel()
```

Returns the level of the planet.

```
planet.isHome()
```

Returns boolean *true* if it is a home planet and *false* if it is a ordinary planet.

```
planet.send(<number>, <to>)
```

Commits a send action of integer *ships* ships, from the planet to planet object *to*. The function returns boolean *true* if successful and *false* on error (e.g. not physically possible/not owner of planet).

```
planet.upgrade()
```

Commits a upgrade command on the planet if it is valid. The function returns boolean *true* if successful and *false* on error (e.g. insufficient ships/not owner of planet).

FLEET OBJECT

IMPORTANT: Fleet objects are the only objects that can be destroyed in the course of the universe. If you retain a fleet object reference, it is your responsibility to check if the reference is still valid by checking it has an owner (i.e. getOwner() is not 0) or if it has any ships (i.e. getShips() is not 0)

`fleet.getOwner()`

Returns an integer *player_id*.

`fleet.getShips()`

Returns the number of ships in the group.

`fleet.getFrom()`

Returns the planet object the fleet was sent from

`fleet.getTo()`

Returns the planet object the fleet is going towards

`fleet.getDistance()`

Returns the distance (in astronomical units) required to travel to reach the destination planet.