Exercise — Creeps

version #1.2

Yet Another Kind of Assistants 2026 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

1 Creeps **4**
1.1 Rush introduction 4
1.2 Game introduction 5

2 Given Files **5**

3 Achievements **6**

4 Hector **6**

5 How to start **7**
5.1 Main program 7
5.2 Viewer 7

6 Game Overview **7**

7 Commands **7**
7.1 Handling 7
7.2 Specifications 9
7.2.1 `noop` 9
7.2.2 `move:<direction>` 9
7.2.3 `observe` 9
7.2.4 `gather` 10
7.2.5 `unload` 10
7.2.6 `farm` 10
7.2.7 `build:<building>` 11
7.2.8 `spawn:<unit>` 11
7.2.9 `dismantle` 11
7.2.10 `upgrade` 11
7.2.11 `refine:<resource>` 12

*https://intra.forge.epita.fr

7.2.12 `message:send <message-parameter>` . . . 12
7.2.13 `message:fetch` . . . 12
7.2.14 `fire:<unit> <fire-parameter>` . . . 12

8 Server Interface **13**
8.1 *Creeps* server . . . 13
8.2 Endpoints . . . 13
8.3 Communicating with the server . . . 14
8.4 GET /status . . . 14
8.5 GET /statistics . . . 14
8.6 POST /init/`username` . . . 15
8.7 POST /command/`login`/`unitId`/`opcode` . . . 15
8.8 GET /report/`reportId` . . . 16

9 Units **17**
9.1 Citizen . . . 17
9.2 Turret . . . 17
9.3 Bomber Bot . . . 17

10 Buildings **17**
10.1 Town Hall . . . 17
10.2 Road . . . 17
10.3 Household . . . 18
10.4 Sawmill / Smeltery . . . 18

11 Behavior and Design Tips **18**
11.1 Units and threading model . . . 18
11.2 Delays . . . 18
11.3 Debugging Threads . . . 19

12 Unit Summary **19**

13 Basic Achievements Summary **20**

File Tree

```
./
├─ .gitignore
├─ README.md
├─ creeps-server.jar
├─ given-javadoc.jar
├─ given.jar
├─ pom.xml
└─ src/
   └─ main/
      └─ java/
         └─ com/
            └─ epita/
               └─ creeps/
                  ├─ **/
                  │  └─ *.java  (to submit)
                  └─ Program.java  (to submit)
```

1 Creeps

1.1 Rush introduction

Creeps is a game where you will have to complete a set of specific actions to obtain achievements. Validation of this activity is based on achievements, **not on traces**. Instead of passing tests, you first have to complete a set of basic achievements to validate the rush and get access to more advanced achievements.

To play the game you will have to interact with a server and implement strategies in order to survive and attain your goals.

1.2 Game introduction

Welcome to *Creeps*! In this exciting and challenging game, you will take on the role of a leader of a small tribe. Your task is to manage resources, expand your territory, and defend against monsters and other players who seek to take what is yours.

The game begins with you and your tribe in a small and vulnerable position. You will need to gather resources such as food and water, build buildings, and create units capable to defend yourself. As you gather resources and strengthen your tribe, you'll be able to expand your territory and claim new land.

But be warned, the wilderness is full of hazards.

Do not just sit back and wait for the attacks: you can also take the initiative and launch your own attacks against rival tribes, with the aim of expanding your territory.

2 Given Files

The server and structure of the project are provided in the architecture described above.

The build system used by this project is Apache Maven. The configuration file, `pom.xml`, is provided. Unless explicitly told by an assistant, do not modify this file.

The entirety of your source code must be placed under the `src/` folder. The entry point of your program must be `com.epita.creeps.Program::main`.

The following libraries are already configured in the provided `pom.xml`:

- Unirest: for REST calls.
- Jackson: for JSON parsing.
- SLF4J + Logback: for logging.
- Validation-api: to add validation contracts to your objects.
- JUnit: for unit testing.
- Lombok: to generate code for minimizing the unused code.
- Given: a library of code containing all DTOs (Data transfer object) and some utility classes to play *Creeps*.

These are the only dependencies allowed for the project.

Documentation about the Given library can be opened in a web browser. This library contains the following:

- `exception`: some exceptions used by the library.
- `Cartographer`: used to register observed tiles by giving it reports. Information about observed tiles can be retrieved in several ways using the given methods. You must use the already initialized INSTANCE object in the class to use these methods.
- Json: utility class to help you serialize your objects to JSON and parse received reports.
- `geometry`: value objects used to describe positions in the map.
- `parameter`: value objects used for command parameters.

- `report`: value objects for command reports.
- `response`: value objects for server responses.
- `Resources`: class representing one's resources.
- `Tile`: enum containing all possible tile types.

Before starting to code anything, make sure you understand all the provided files. We **strongly** advise you to read the `README.md`, the given documentation and the subject at least twice.

3 Achievements

Validation is based on achievements, **not** on traces. You will have to validate the basic achievements to validate the rush. Keep in mind that validating only the basic tree will only give you enough points to validate the rush. To improve your grade, you will have to complete more advanced achievements.

Achievements will only count as completed if achieved on the live game server. You can push as many times as you want, and **previously completed achievements will not need to be redone**.

To get the list of all achievements and how to complete them, add the `--printAchievements=true` parameter when starting the server.

Achievements need to be activated with `--trackAchievements=true`.

4 Hector

Hector is the "space-cleaner", hence the saying: "When you've got garbage, call Hector".

When Hector is enabled in the server options, he will perform a GC (Garbage Collection) pass at fixed time intervals, similarly to how a garbage collector works in programming languages.

During a GC pass, Hector will try to find reachable buildings. To do so, Hector will browse space starting at townhalls. Following built paths, Hector will mark all reachable buildings. Every building that is not reachable is considered "unreachable". Every unit not standing on a reachable building will also be considered "unreachable".

Once the mark phase is done, Hector will process to the collection phase, in which **he destroys all unreachable buildings and kills every unreachable unit**.

The first GC pass is done at tick 0. After that, Hector will perform a GC pass at a given tick rate, set in the server configuration.

5 How to start

5.1 Main program

You have to declare a main function in the Program class. Your program takes 3 arguments: the server's URL, the server's port, and your login. When using the local server, the URL will be `localhost` and the port will be `1664` by default.

Pass these values to your program – do not hardcode them in your code, or we will be unable to run your submissions.

5.2 Viewer

To help you visualize what is happening during a game, you can connect to your server using any modern web browser as long as `--enableWebClient` option is set to true (which is the default). Whenever your server is running, you just have to connect to http://localhost:port, where the port is indicated by the value of `--httpPort` (default is 1337).

You can view any player's game by clicking on them on the right-hand side panel, as well as check their achievements. You can also move the map using the 4 buttons to move up, down, left or right. You will also be notified when the viewed player completes an achievement.

6 Game Overview

At the beginning of a game, you will be provided with a townhall and two `citizens`, placed in a 2-dimensional world composed of tiles. The map has no border. Those citizens can execute commands like creating new buildings, harvesting resources or spawning new units. Managing your resources will not be the only problem on your hands, as monster and other players will try to attack you too. To play the game efficiently you will have to manage your resources wisely and defend your territory, or maybe go for a more offensive strategy an attack other territory.

There are several ways to reach the end of a game. The game stops when you have no citizen or no town hall left.

7 Commands

7.1 Handling

Each command has a casting time in ticks, and its call might also cost some resources. This cost information is provided to the player when they log in, in the init response.

When a player submits a command, the following process happens:

1. The game performs some safety checks on the command, that can lead it to being rejected. In this case, you will synchronously receive an error report as described in the command and report API section below.

The safety checks cover everything that can be verified upfront:

- Is the game running?
- Is the opcode valid?
- Is the player valid and alive?
- Is the unit valid and alive?
- Is the unit available (i.e., not already casting)?
- Can the player afford the resources for the command?

2. The command is scheduled to execute after its casting time has passed. At the moment it is scheduled, a response is given containing the report id that must be used to fetch another report when the action will have finished executing.
3. Once the scheduled time is reached, the command is executed.
4. The command can result in either success or failure for a variety of reasons (for example, the unit might be killed while casting). If the command results in a success, the necessary resources are removed from the player's inventory. If the command results in a failure, an error code describing the nature of the failure is given in the report.
5. Since the command is performed in an asynchronous fashion, the game cannot notify you directly. Instead, a report in JSON format is produced, which you can readily query and consult.
6. The unit is made available again.

All commands generate a report, and all reports will contain at least the following information:

```
{
    "opcode" : "status",          // Command opcode
    "reportId" : "1a2b3c4d5",     // Report ID
    "unitId" : "a1b2c3d4e",       // Unit ID
    "login" : "xavier.login",     // Player login
    "unitPosition" : {...},       // Position of the unit
    "status" : "SUCCESS"          // Execution status, SUCCESS or ERROR
    "tick" : 42                   // The tick at which the command took place
}
```

An `errorCode` field will appear in error reports, with a specific code to indicate what went wrong.

Here are some general error codes:

- `unit-dead`: the unit is dead.
- `player-dead`: the unit's player is dead.
- `unit-of-another-player`: the unit belongs to another player.
- `operation-invalid`: the command is not handled by the unit.
- `unit-unavailable`: the unit is already casting a command.

Some commands may add more information, which will be described below.

7.2 Specifications

7.2.1 `noop`

Does nothing. Useful mainly for testing and status purpose.

7.2.2 `move:<direction>`

Moves the agent according to the direction suffix. This suffix can either be `up` (y+), `down` (y-), `right` (x+) or `left` (x-).

Example: `move:left`

When moving, a unit will also gather information about the tiles and units around it in a square of 5x5 tiles.

Adds three fields to the standard report:

- `newPosition`: the new position of the unit.
- `tiles`: the list of observed tiles.
- `units`: the list of observed units.

Each element in the `units` field follows this format:

```
{
    "player" : "xavier.login",  // Player login
    "opcode" : "turret",        // The unit's opcode
    "position" : {...},         // The unit's position
}
```

The report can be given to the Cartographer to register the observed tiles.

7.2.3 `observe`

Gives information about the tiles and units around a unit, in a square of 6x6 tiles.

Adds two fields to the standard report:

- `blocks`: the list of observed tiles.
- `units`: the list of observed units.

See the `move` command above for details about the fields for a unit.

The report can be given to the Cartographer to register the observed tiles.

7.2.4 `gather`

Gathers a resource from the tile the unit is standing on.

Citizens have a fixed inventory size as well as fixed gathering rate for each resource, which are given in the init response. The inventory can gather any type of resource, at once, but the sum of all resources cannot exceed the inventory size.

If a citizen cannot carry the gathered resource, the inventory is filled and the remaining quantity is lost. If the gathering rate is greater than the resource quantity, the amount inserted into the inventory is only the resource quantity.

Adds three fields to the standard report:

- `resource`: the opcode of the gathered resource.
- `gathered`: the quantity of gathered resources.
- `resourcesLeft`: the quantity of resource left on the tile.

The report can be given to the Cartographer to update the resource tile.

Possible error codes: `not-resource-tile`.

7.2.5 `unload`

To benefit from the gathered resources, you will have to make the citizen unload its inventory at a town hall. To do so, simply move the citizen to any of your town halls and cast the `unload` command on that citizen.

Adds one field to the standard report:

- `creditedResources`: the resources added to the player's resources.

Possible error codes: `not-on-town-hall`.

7.2.6 `farm`

Grows a food tile where the citizen is currently standing. Farming is only allowed on empty tiles adjacent to water tiles.

Adds one field to the standard report:

- `foodQuantity`: the quantity of food created.

The report can be given to the Cartographer to register the newly created food tile.

Possible error codes: `tile-occupied`, `no-water-nearby`.

7.2.7 `build:<building>`

Builds the specified structure on the tile the unit is currently standing on. The tile must be empty to build on it. Beware of Hector. More information about the buildings is available in the buildings section.

The possible buildings are: `town-hall`, `household`, `sawmill`, `smeltery` and `road`.

Adds one field to the standard report:

- `building`: the newly created building.

Additionally, if the building is a household, another two fields are added to the report:

- `spawnedCitizen1Id`: the ID of the first spawned citizen.
- `spawnedCitizen2Id`: the ID of the second spawned citizen.

The report can be given to the Cartographer to register the newly created building.

Possible error codes: `tile-occupied`, `insufficient-funds`.

7.2.8 `spawn:<unit>`

Spawns the given mechanical unit at the spot it is invoked on. Mechanical units are defined as either a `turret` or a `bomber-bot`.

Adds two fields to the standard report:

- `spawnedUnitId`: the ID of the new unit.
- `spawnedUnit`: the spawned unit.

See the `move` command for details about the fields for a unit.

Possible error codes: `insufficient-funds`.

7.2.9 `dismantle`

Removes from the game the mechanical unit who casted the command.

7.2.10 `upgrade`

Upgrades the unit. Upgraded units have their cast time halved (rounded down). If the unit is a citizen, it can see further with the `observe` command. If the unit is a turret, its range is increased by two. If the unit is a bomber bot, its range and blast radius is increased by one.

Every upgraded citizen needs 2 units of food instead of 1 every time citizens are fed.

Possible error codes: `unit-already-upgraded`, `insufficient-funds`.

7.2.11 `refine:<resource>`

Refines resources to create the specified refined resource. A refined resource can either be `copper` or `wood-plank`. Refined resources are used to upgrade units.

Possible error codes: `not-on-suitable-refinery`, `insufficient-funds`.

7.2.12 `message:send <message-parameter>`

Sends a message of your choice to another player. A message cannot exceed 100 characters.

Adds one field to the standard report:

- `recipient`: the recipient of the message.

The `message-parameter` class is available in the given files and contains two fields:

- `recipient`: the recipient of the message.
- `message`: the message to send.

This parameter must be serialized and passed to the body when posting your command.

Possible error codes: `message-to-self`, `message-too-long`.

7.2.13 `message:fetch`

Returns the list of all messages received since the last `fetch` call. Can only fetch up to the last 50 received messages.

Adds one field to the standard report:

- `fetchedMessages`: the list of received messages.

Each message in the `fetchedMessages` field follows this format:

```
{
    "sender" : "xavier.login",  // The player who sent the message
    "message" : "Hello :)",     // The message's content
}
```

7.2.14 `fire:<unit> <fire-parameter>`

Fire from the target unit to a specified destination tile. The unit can either be `turret` or `bomber-bot`.

A `turret` has a base range of five tiles and can only kill units on the targeted position. A `turret` cannot target its own position.

A `bomber-bot` has a base range of two tiles and a blast radius equal to its range. Therefore, a `bomber-bot` will always kill itself when firing. The `bomber-bot` is the only unit capable of destroying raider camps when using the `fire` command.

Adds two fields to the standard report:

- `target`: the targeted position.
- `killedUnits`: the list of units, friendly and enemy, killed by the shot.

See the `move` command for details about the fields for a unit.

The `fire-parameter` class is available in the given files and contains one field:

- `destination`: the targeted position.

This parameter must be serialized and passed to the body when posting your command.

Possible error codes: `out-of-range`, `turret-minimum-range`.

8 Server Interface

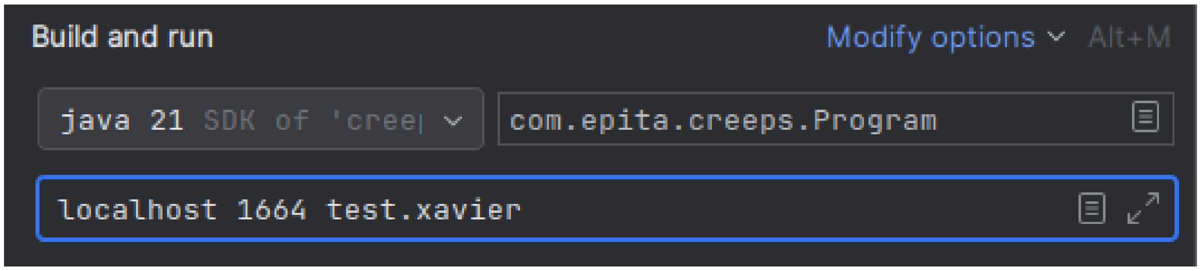
8.1 *Creeps* server

In order to launch your program on the right IP and port, your program will be executed as follow:

```
java -jar target/creeps-1.0.0-SNAPSHOT.jar [HOSTNAME] [PORT] [USERNAME]
```

When using the local server, hostname will be "localhost" and port will be 1664. To unlock achievements, you need to be connected to a public server. The server will not immediately be available so you can first test your automatons locally.

To launch your program on IntelliJ, you can use the following configuration:



```
java 21 SDK of 'cree
com.epita.creeps.Program
localhost 1664 test.xavier
```

8.2 Endpoints

A server exposes 5 HTTP endpoints through which you can interact:

- GET /status
- GET /statistics
- POST /init/`login`
- POST /command/`login`/`unitId`/`opcode`
- GET /report/`reportId`

In order to play the game, you will have to make HTTP requests to the server. You will have to use both GET and POST requests. As a reminder, a GET request is used for viewing a resource, while a POST request is used for changing a resource.

In the latter case, you may have to create a body, which will contain some information about what you want to modify. To be understood correctly by the server, all your bodies must be in a *JSON* format.

For this rush, you are allowed to use Unirest with the Jackson library.

8.3 Communicating with the server

```
// Send a GET request
Unirest.get(uri).asJson()
// Send a POST method
Unirest.post(uri).body("{}").asJson()
```

You will need to read the documentation of those two libraries to have a better understanding of how to use them.

8.4 GET /status

Returns a *JSON* object with a single `running` field indicating whether or not the game is still running.

```
{
  "running": false
}
```

8.5 GET /statistics

Returns the current statistics of the server.

```
{
  "serverId": "Pop",
  "gameRunning": false,
  "tick": 42
  "dimensions": {
    "x": 10000,
    "y": 10000
  }
  "players": [
    {
      "name": "xavier.login",
      "status": "ALIVE",
      "units": 8,
      "buildings": 3,
      "resources": {
        "rock": 30,
        "wood": 30,
        "food": 30,
        "oil": 0,
        "copper": 0,
        "woodPlank": 0,
      }
```

(continues on next page)

(continued from previous page)

```
      "achievements": ["Hello, World!"]
    }
  ]
}
```

8.6 POST /init/`username`

Allows you to connect to the game.

Once you are connected, you are provided with all the information you need to start playing. Namely:

- Your player ID, used to differentiate between players' buildings.
- The position of your town hall in the map.
- The position of your household in the map.
- The ID of your two citizens currently standing on your household.
- A summary of the cost of all commands available in the game.
- Your starting resources.
- The server's configuration.

If the `error` field is not `null`, it contains one of the following error messages, describing what error occurred:

- `"login unavailable"`

  The `login` you provided is not available, try again with another one.

- `"new players are not allowed"`

  The server is about to restart and new players cannot join.

For more information, take a look at the `InitResponse` class of the Given library.

8.7 POST /command/`login`/`unitId`/`opcode`

Orders the unit with the given `unitId` to perform the command with the given `opcode`.

```
{
  "opcode" : "command",       // Information about command success
  "reportId" : "1a2b3c4d5",   // Incoming report ID
  "errorCode" : null,         // Error description, might be omitted when null
  "error" : null,             // Full error, might be omitted when null
  "login" : "",               // Player login, might be empty
  "unitId" : "",              // Unit ID, might be empty
  "misses" : 0                // Number of misses
}
```

If the `errorCode` field is not `null`, one of the following errors occurred:

- `"notrunning"`

  The game is not running.

- `"unrecognized"`

  The `opcode` you requested was not recognized.

- `"noplayer"`

  The `login` you provided did not match any player on the server.

- `"unavailable"`

  Your unit is already doing something. Wait until it finishes before assigning it another job. Note that your missed calls counter has been incremented. If it goes over a certain value, the next missed call will lead to the death of the corresponding unit.

- `"noresources"`

  You do not own enough resources at the moment. Try again later when you do.

- `"dead"`

  Your unit died. Note that that report is only sent once, after which you will receive a “nounit” response.

- `"nounit"`

  The `unitId` you provided did not match any of your units.

- `"invalidparameter"`

  The provided parameter does not match the given command. It might not have been serialized properly.

8.8 GET /report/`reportId`

Retrieves the report with the given `reportId`.

You will find the structure of the response of each possible opcode in the given library.

The report is only available after the duration specified for each task has passed. If `reportId` does not exist, or if the report is not yet ready, you will get the following response:

```
{
  "opcode" : "noreport",
  "error" : "No such report",
  "reportId" : "173040eba"
}
```

9 Units

This section describes all types of units available to you. Units are meant to be told what to do using commands, which are detailed in the section above.

9.1 Citizen

The citizen is the most important unit in the game. They are very versatile and will carry out any task that does not involve attacking of defending yourself. You start the game with 2 citizens. Every household you build will spawn two citizens.

9.2 Turret

Turrets are powerful defensive units but cannot move. They are the core of your defense but can also kill your own units, so be careful. A turret can be spawned by any citizen.

9.3 Bomber Bot

Bomber bots are an offensive type of unit. Their attack is short-ranged, but in a wide area, and they are the only way to destroy raider camps. However, after they attack, they immediately die. A bomber bot can be spawned by any citizen.

10 Buildings

Buildings are structures built by citizens. Buildings have a wide range of uses, but are more passive gameplay elements compared to units.

10.1 Town Hall

The town hall is the main building of your village. You will lose if all your town halls are destroyed, so you must protect them at all costs. Town halls are the only building safe from Hector, and every other building must be connected to a town hall to be protected as well.

10.2 Road

Roads are very cheap and allow you to expand your territory to protect yourself from Hector.

10.3 Household

The household will spawn two citizens upon being built.

10.4 Sawmill / Smeltery

The sawmill and smeltery are refineries. They will respectively be used to refine wood planks and copper. A unit needs to stand on the correct refinery to be able to refine resources.

11 Behavior and Design Tips

11.1 Units and threading model

Even though it would be possible to implement an AI over a single execution thread, said AI would be very limited in terms of its capabilities. We advise you to adopt a more advanced design wherein all your units could be dispatched over not one but several threads. This would allow you to scale up to dozens or even thousands of units on your computer depending on your implementation.

As you were told during the presentation of this project, we encourage you to gather information about the following subjects:

- BlockingQueue
- CompletableFuture
- Threading in *Java*

As most of these notions were covered during the courses (TDs), your first step should be to have a second – if not first – look at those.

11.2 Delays

As you probably know by now, a unit cannot just chain its commands. Each command takes a certain number of ticks to be executed (a delay), meaning that a unit can only execute a command once the delay of its previous command has passed. You are **not allowed** to use `Thread.sleep()` to make a unit wait between commands. It is up to you to find a smarter way to handle delays.

All durations are given in ticks. The number of ticks per second is communicated by the server in the init response.

11.3 Debugging Threads

Printing to standard output is a bad idea, as you will probably be flooded by messages, and you will not be warned of thread crashes. Using the debugger will work, provided that the right thread is caught.

You are strongly advised to use a `Logger` for each of your units. The principle is simple: each unit contains a `Logger` instance that will write messages to a file with a specific name. This will allow you to follow the behavior of your units individually.

12 Unit Summary

Note that the given durations are here to give you a general idea of how long actions take. Use the init response to get the actual durations for every command.

Opcodes	**Citizen**	**Turret**	**Bomber Bot**	**Cartographer**	**Duration**
noop	yes	yes	yes	no	1
move:*	yes	no	yes	yes	2
observe	yes	no	no	yes	1
gather	yes	no	no	yes	4
unload	yes	no	no	no	3
farm	yes	no	no	yes	10
build:*	yes	no	no	yes	6-20
spawn:*	yes	no	no	no	6
dismantle	no	yes	yes	no	1
upgrade	yes	yes	yes	no	1
refine:*	yes	no	no	no	8
message:*	yes	no	no	no	1
fire:turret	no	yes	no	no	2
fire:bomber-bot	no	no	yes	no	6

13 Basic Achievements Summary

Note that achievements are categorized into 6 groups: Basic, PVP, Spatial / PVE, Communicationand, Fun, and Economy.

Achievements	**Description**
BobTheBuilder	Build any building.
Clean Memory	Survive one GC cycle without losing any unit or building.
Dead Cells	Die.
FireAtWill	Fire with any capable unit.
Gather a resource	Gather a resource.
Good Player!	Play a game with at least 25 cycles and send 50 commands without missing any.
Hello, World!	Connect to the game.
It's Aliiiiivvvve!	Spawn any mechanical unit.
Let's go!	Send a valid command.
Oscar Mike	Move any unit.

Being a hero means fighting back even when it seems impossible.