Advanced Java 2016
Final exam
Monday, 29 August, 9:00 — Tuesday, 30 August 23:59

This exam set consists of two parts:

1. A problem description on 5 numbered pages, including this page. Make sure you have them all, and read this page first.
2. A zip-archive containing a stub implementation of the exam.

Format. The exam has two parts. Each part contains a number of questions which you must solve by filling out the missing parts of the stub implementation. Your only deliverable is the solution code, no report is required. *Please make sure to comment your code appropriately.*

Clarifications. In the event of errors or ambiguities in the exam text, **do not use the discussion forum**. Instead, please contact one of Danil (daan@di.ku.dk), or Vivek (bonii@diku.dk) by email. Do not expect an answer to the email; clarifications will be published on the course web page. Clarifying updates are posted at regular intervals at **Monday 13:00**, **Monday 17:00**, **Tuesday 9:00**, **Tuesday 13:00**. These postings are considered part of the exam text, and it is your responsibility to check the course web page.

Hand in. You must submit your solution via the Absalon course page under the header "Exam 2016". Your submission must adhere to the following requirements:

- It must consist of a single zip-file containing your solution code. You can omit submitting the lib/ directory handed out in zip archive of the exam.
- All files must be able to compile (i.e., be accepted by the Java 8 compiler without errors), and any code required for building must be included in the submission.

The deadline at the top of this page is strict. You may submit multiple times, the *last* submission counts and the previous will be ignored.

Exam fraud. Your solution to this take-home exam must be generated by yourself, *completely individually*. For the entire duration between the release of this exam and the deadline at the top of this page, you are not to communicate with any other person about this exam (whether helping or receiving help), except for the course instructors. Both submitting solutions that are not wholly your own or sharing your own solutions with other students are considered exam fraud. Any violations will be handled in accordance with Faculy of Science disciplinary procedures.

Happy exam!

Background - Pokemon Adventure

Pokemon Adventure is a fictitious fantasy game modeling a simplified Pokemon world. The Pokemon world consists of 3 types of game entities, (1) Pokemons, (2) Players and (3) World changers. In this game, Pokemons are creatures that randomly move around in the Pokemon world. Players try to kill a Pokemon or bring a dead Pokemon back to life depending on their roles. Pokemon world changers are housekeepers of the Pokemon world and can create/destroy players and Pokemons, thus adding/removing them from the Pokemon world.

This exam is about implementing the game entities and their behavior using a web service thus simulating the Pokemon world in action!

The Pokemon World Manager

The Pokemon world is an unbounded two dimensional world modeled using Cartesian coordinate system. An entity in the Pokemon world can reside in any valid location subject to the constraint that *no other entity* whether dead/alive is occupying that location. Every game entity in the Pokemon world is assigned a unique identifier(id). The Pokemon world consists of the following entities:

1. Pokemons - Pokemons are native, fun-loving creatures of the Pokemon world. Each Pokemon is an autonomous entity and must have an independent thread of execution. A Pokemon freely moves around the Pokemon world by randomly choosing a location to move to and being present in that location for some amount of time which can vary for different Pokemons. A Pokemon can be killed by hunters and can be brought back to life by rejuvenators.

2. Hunters - Hunters are controlled by players in the game. Their objective is to hunt Pokemons. In order to hunt a Pokemon, they must know the location of the Pokemon, the unique id of the Pokemon and must be within visible range of the Pokemon. Their movements are controlled by user who specifies the location that the hunter can move to.

3. Rejuvenators - Rejuvenators are also controlled by players in the game. Their objective is to bring dead Pokemons back to life. In order to bring a dead Pokemon back to life, they must know the location of the dead Pokemon, unique id of the Pokemon and must be within visible range of the Pokemon. Their movements are controlled by the user who specifies the location that the rejuvenator can move to.

4. World changers - World changers are the maintainers of the Pokemon world. They do not reside in the Pokemon world. They can create and destroy Pokemons, hunters and rejuvenators. Once they destroy a game entity, it must be removed from the Pokemon world. In order to destroy any entity, knowing the id of the entity is sufficient for the world changers.

In order to simplify the game, we provided some physical relaxations in the Pokemon world. A game entity (Pokemon/hunter/rejuvenator) can move to any valid location in the game without being constrained by speed of movement. Every entity in the game has the same visibility. Visibility of an entity is defined by the grid formed using the entity's current location as the centre of the grid and the visibility parameter acting as the bounds of the grid in both dimensions (X and Y). An entity in the game can see every entity (dead/alive) within the visible range of a location. A game entity is not required to move to a location in order to see the game entities visible around that location. A game entity can request the visible entities around any location in the Pokemon world.

The Pokemon world manager maintains the Pokemon world and is implemented in the package `advjava.exam16.model`. Important classes include:

`APokemon` Represents a Pokemon in the Pokemon world. This class implements the Runnable interface in order to model each Pokemon as an independent thread of execution. Once a Pokemon randomly picks the location that it must move to, it must try to move to that location by requesting the Pokemon world manager. The time that a Pokemon must spend at a location before moving to another is configured using the constructor. `APokemon` must also support thread termination/suspension in order to support killing of a Pokemon by a hunter.

`AWorldManager` Represents a concrete implementation of Pokemon world manager which maintains a consistent view of the Pokemon world. It must encapsulate the Pokemon world. All the game entities must interact with this class in order to read and update the Pokemon world.

`ImmutablePokemonWorldEntity` Represents an immutable game entity (Pokemon/hunter/rejuvenator) in the Pokemon world. This immutable entity is used by the Pokemon world manager to allow game entities to read the Pokemon world safely. You are allowed to implement your own mutable implementation to represent entities in the Pokemon world inside `AWorldManager` class.

`IntegerLocation2D` Represents a location in the Pokemon world.

The Pokemon world manager `AWorldManager` must implement the methods defined in the package `advjava.exam16.interfaces`. Important classes in the package include:

`WorldViewManager` Interface that defines the methods that all game entities can invoke to get all visible entities (alive/dead) near a location in the Pokemon world.

`WorldViewChanger` Interface that defines the methods that only world changers can invoke to create/destroy entities in the Pokemon world. Creation of an entity must assign it a unique identifier in the Pokemon world. Once an entity is destroyed, it must be removed permanently from the Pokemon world. It also extends the `WorldViewManager` interface.

`PokemonWorldManager` Interface that defines the methods that Pokemons can invoke to move around in the Pokemon world.

`HunterWorldManager` Interface that defines the methods that hunters can invoke to move around in the Pokemon world and to kill a Pokemon.

`RejuvenatorWorldManager` Interface that defines the methods that rejuvenators can invoke to move around in the Pokemon world and to bring a dead Pokemon back to life.

`IWorldConstants` Constants defined for the Pokemon world. The parameter "WORLD_VISIBILITY" defines the size of visibility grid around a location. All game entities entities have the same visibility. You are allowed to add more constants to this file and modify the value of the "WORLD_VISIBILITY" parameter.

The Pokemon World Service

The Pokemon world is maintained on a server by the Pokemon world manager (`AWorldManager`) including the individual Pokemon objects (`APokemon`) which are modeled as independent, concurrently running threads on the server. We should be able to provide an HTTP interface for the game players (hunters/rejuvenators) and world changers which are modeled as clients for requesting operations to read and update the Pokemon world on the server. The interfaces for individual game clients must reflect the interfaces defined above to enforce the operations that different game entities are allowed to perform on the Pokemon world.

A Jetty-based web service is implemented in the package `advjava.exam16.service`. Important classes and interfaces include:

`HTTPHunterClientService` Implementation of `HunterWorldManager` which uses an HTTP client to allow communication of hunter clients with a remote Pokemon world manager.

`HTTPRejuvenatorClientService` Implementation of `RejuvenatorWorldManager` which uses an HTTP client to allow communication of rejuvenator clients with a remote Pokemon world manager.

`HTTPWorldChangerClientService` Implementation of `WorldViewChanger` which uses an HTTP client to allow communication of world changer clients with a remote Pokemon world manager.

`RequestHandler` An HTTP request handler which exposes a web interface to any `AWorldManager` implementation.

Part 1 - Pokemon World Manager

Task 1

Read all the classes in the `advjava.exam16.model` package. Finish the implementation:

1. Finish the implementation of `ImmutablePokemonWorldEntity`.
   a) Implement `equals`. This must indicate whether some object is equal to "this" one.
   b) Implement `hashCode`. This must return the hash code value for the object.
2. Finish the implementation of `APokemon`.
   a) Implement `APokemon` constructor. This should initialize the internal state of the object. The parameter "locationStayingTimeInMSecs" determines how long a Pokemon must spend at a location before it moves to another location in the Pokemon world.
   b) Implement `run`. This should be entry point of the Pokemon thread. The thread should run continuously and try to randomly move to a new location after spending the configured amount of time at a location. You should not use busy waiting to simulate staying at a location (Hint: Try suspending/sleeping).
   c) Implement `kill`. This should either terminate/suspend the Pokemon thread. The method should block until termination/suspension of the thread is complete.
3. Finish the implementation of `AWorldManager`, which encapsulates and manages the Pokemon world.
   a) Implement `getNearbyEntities` which takes a location as parameter. This should return all the entities (dead/alive) within the visibility range of the location parameter.
   b) Implement `createEntity` which takes a location and an entity type as parameter. This should create the entity of the requested type at the requested location and assign it a randomly generated name and a unique identifier. The unique identifier of the created entity is returned. Appropriate errors must be signaled. (See Javadoc)
   c) Implement `destroyEntities` which takes a set of entity identifiers as parameter. This should remove all the entities represented by the list of identifiers passed as parameter from the Pokemon world. Appropriate errors must be signaled. (See Javadoc)
   d) Implement `killPokemon` which takes a Pokemon identifier, a hunter identifier and a Pokemon location as input. If the Pokemon identified by the identifier is alive at the location passed as parameter and is within the visibility range of the hunter, then the Pokemon must be killed by invoking `kill` in `APokemon` class. The Pokemon thread can either be suspended/terminated and the method must only return when suspension/termination is complete. The Pokemon must not be removed from the Pokemon world but must be marked dead at the location. Appropriate errors must be signaled. (See Javadoc)
   e) Implement `moveHunter` which takes a hunter identifier and a target location as input. If the target location is not occupied by another game entity, then the hunter must be moved to the target location. Appropriate errors must be signaled. (See Javadoc)
   f) Implement `rejuvenatePokemon` which takes a Pokemon identifier, a rejuvenator identifier and a Pokemon location as input. If the Pokemon identified by the identifier is dead at the location passed as parameter and is within the visibility range of the rejuvenator, then the Pokemon must be brought back to life. Depending on the implementation employed to kill a Pokemon, either a new thread must be created or the suspended thread must be made to run again before the method returns. Appropriate errors must be signaled. (See Javadoc)
   g) Implement `moveRejuvenator` which takes a rejuvenator identifier and a target location as input. If the target location is not occupied by another game entity, then the rejuvenator must be moved to the target location. Appropriate errors must be signaled. (See Javadoc)
   h) Implement `movePokemon` which takes a Pokemon identifier and a target location as input. If the target location is not occupied by another game entity, then the Pokemon must be moved to the target location. This method is invoked by the Pokemon thread to change its location. Appropriate errors must be signaled. (See Javadoc)

Task 2

Test your implementation using JUnit. In this task, you need to figure out what are the interesting test cases. You must implement your JUnit test cases in `PokemonTest` class inside `advjava.exam16.tests` package.

1. Test that your implementation of `ImmutablePokemonWorldEntity` satisfies the requirements of `equals` and `hashCode`.
2. Test that your implementation of `AWorldManager` works as expected. That is, test the functional properties of the methods defined.

Part 2 - Pokemon World Service

Note: This part depends on the Pokemon world manager from Part 1.
Finish the implementation of the Pokemon world manager service architecture.

1. Implement the `HTTPHunterClientService` class. This is an implementation of the Pokemon world manager for the hunter which forwards all calls to a remote Pokemon world manager service.
    a) Implement the `HunterWorldManager` interface. Exceptions from the service should be re-thrown in the client.
2. Implement the `HTTPRejuvenatorClientService` class. This is an implementation of the Pokemon world manager for the rejuvenator which forwards all calls to a remote Pokemon world manager service.
    a) Implement the `RejuvenatorWorldManager` interface. Exceptions from the service should be re-thrown in the client.
3. Implement the `HTTPWorldChangerClientService` class. This is an implementation of the Pokemon world manager for the world changer which forwards all calls to a remote Pokemon world manager service.
    a) Implement the `WorldViewChanger` interface. Exceptions from the service should be re-thrown in the client.
4. Implement `RequestHandler`.
    a) Implement the `handle` method to handle incoming HTTP requests and dispatch them to an underlying `AWorldManager` object. Your implementation should propagate exceptions thrown by methods in the `AWorldManager` implementation to the client.

For debugging your implementation, you can use the class `HTTPServerApplication` to start a Pokemon world management service server via Jetty, and the class `HTTPClientApplication` for starting a client. The client connects to a server and creates hunters, rejuvenators and Pokemons at random location. It then tries a combination of operations to move game entities, kill Pokemons, rejuvenate Pokemons and finally destroys all the created game entities.