Introduction

Now that the first assignment is done, it is time to move on to a *bigger challenge*. For this assignment you will have three weeks; including an optional demo. Be sure to start early, as this is a large assignment! In this assignment you will make use of threads and TCP to simulate chat servers.

All the material you need can be found in the lectures and reader. Additionally, there will also be tutorials to demonstrate some of the concepts live.
Be sure to read through the entire assignment thoroughly!

Deadlines

The assignment is split into two parts. Similarly there are two deadlines:

- **Deadline 1: 24th of September at 23:59**
  For this deadline, part 1 should to be finished. If there are a few minor things missing, this is not a very big deal, but we want to see you make proper progress. This will not be graded, but if we find that you have not made sufficient progress, it might result in a penalty of at most -1 point. While the demo for this deadline is optional, creating a pull request is mandatory!
- **Deadline 2: 1st of October at 23:59**
  This is the final deadline. This means that both part 1 and part 2 should be finished. Both creating a pull request and attending the demo is mandatory!

Chatrooms

Recently the RUG has begun constructing a new building. When construction began, they accidentally stumbled upon an ancient server park. Experts have dated it as far back as 2013. After a lot of research, they have discovered that it used to run a magnificent chat server software called Enrage. They found that is was initially constructed to study the curious behaviour people show when talking to strangers. The project was terminated after a few months as the developers spent too much time thinking of funny chat messages for the bots.

Now, however, the university has once again gained an interest in reviving this old simulation. Lucky for us they happened to specifically request that it be the second assignment for Advanced Object-Oriented Programming. Due to this amazing coincidence it is your job to construct this simulation.

Components

There are four major components to this simulation:

- A main server
- Chatrooms
- BotManager
- Bots

The code you will create for this simulation will not run as one application, but instead as a lot of independent processes. The main server, each chatroom and the BotManager should be their own processes. This means that these components are going to need their own `main()` methods.
We will explain later on how to easily run a lot of processes using IntelliJ.
All the separate processes should communicate via networking. For this assignment, you will be using TCP to achieve this communication.

Before you start, we highly advise you to first create a drawing of what your class structure will look like. This will make it easier to properly structure your project.

Part 1: Main server & Chatrooms

The goal of part 1 is to implement the main server and the chatrooms.

Main server

The process that needs to be started first is the main server. The main server keeps track of all the chat rooms and is responsible for distributing port numbers to the bots. For this reason, the only port number that you are allowed to "hardcode" in your application is that of the main server. The chatrooms (and bots) should be aware of this port number and will always first connect to the main server.

Chatrooms will inform the main server of the port number they are listening on. The main server will then register this port number as an active chatroom. This list of port numbers is used later on to provide the bots with the port numbers of these active chatrooms.

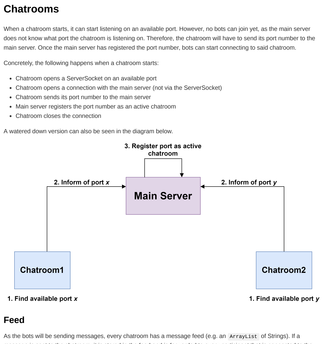
Chatrooms

When a chatroom starts, it can start listening on an available port. However, no bots can join yet, as the main server does not know what port the chatroom is listening on. Therefore, the chatroom will have to send its port number to the main server. Once the main server has registered the port number, bots can start connecting to said chatroom.

Concretely, the following happens when a chatroom starts:

- Chatroom opens a ServerSocket on an available port
- Chatroom opens a connection with the main server (not via the ServerSocket)
- Chatroom sends its port number to the main server
- Main server registers the port number as an active chatroom
- Chatroom closes the connection

A watered down version can also be seen in the diagram below.



Feed

As the bots will be sending messages, every chatroom has a message feed (e.g. an `ArrayList` of Strings). If a message is sent to the chatroom, it is stored in the feed and is forwarded to every participant that is connected to the chatroom. This way each connected bot is aware of the messages that are being posted. As a result, each bot essentially has its own local copy of the message feed.

Part 2: Bots

Since we want our chatrooms to be the opposite of empty, we are going to need some bots.
The application should support two types of bots:

- **Local bots**
  A local bot requests the port of a random active chatroom once and then remains in that chatroom; it does not leave.
- **Migratory bots**
  A *migratory* bot can switch between chatrooms. Each time this bot sends a message, there is a chance that it "migrates" to a different server instead. It can stay in a chatroom for a while and then switch to a different chatroom. Switching chatrooms happens via the main server. It will leave the current chatroom and send a request to the main server again. The server will send back a port number of a *random* (but different!) active chatroom. Once the bot has received this new port number, the bot will connect to the corresponding chatroom.

Bot behaviour

A bot should send a chat message every few seconds. For this you could pick a random number of seconds between 2 and 7 (feel free to change these numbers). However, before it sends each message, the bot should determine what it is going to send. This depends on the current message feed.

Of course we want our bots to do some fun stuff, so bots can create these messages in different ways. We will refer to this as the behaviour of a bot. Two simple examples of behaviours could be:

- Echo the last message in the feed.
- Take the last *x* messages, combine all the words and pick a random selection of those words.

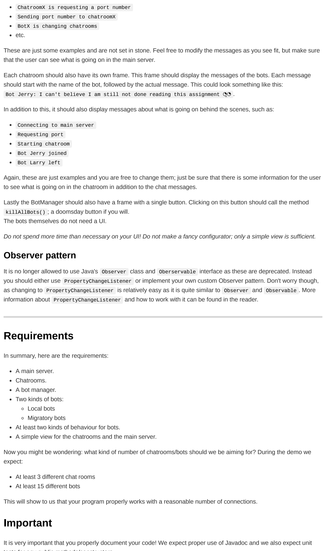
You are encouraged to think of more ways to create new fancy messages!
The behaviour that a bot has is not dependent on the type of bot it is. Instead, a behaviour is assigned randomly when the bot is created. It is up to you to find a proper Object-Oriented way of implementing these behaviours.

Lastly, to give your bots a bit more personality, feel free to give them names or other cool features.

BotManager

The BotManager is responsible for spawning all of the bots. As the bots should be independent, each bot will need its own thread.
Other than creating the bots, the BotManager should also have a method `killAllBots()` that kills all the bots (surprise). This could, for example, be achieved by ensuring that the boolean value that controls the `while()` loop in the bot's `run()` method is set to `false`.
Once this method is called, no new messages should appear in the chatrooms.

> Hint: as you will be spawning a bunch of threads, you are advised to use Java's `ExecutorService`.



View

We are obviously very curious to see what kind of fancy spam the bots create. Since multiple threads printing to a single console is very difficult track, make a **simple** UI. This UI should only consists of frames that display print statements.

The main server should have such a frame. In this frame, messages could be displayed such as:

- `ChatroomX is requesting a port number`
- `Sending port number to chatroomX`
- `BotX is changing chatrooms`
- etc.

These are just some examples and are not set in stone. Feel free to modify the messages as you see fit, but make sure that the user can see what is going on in the main server.

Each chatroom should also have its own frame. This frame should display the messages of the bots. Each message should start with the name of the bot, followed by the actual message. This could look something like this:
`Bot Jerry: I can't believe I am still not done reading this assignment >>`.

In addition to this, it should also display messages about what is going on behind the scenes, such as:

- `Connecting to main server`
- `Requesting port`
- `Starting chatroom`
- `Bot Jerry joined`
- `Bot Larry left`

Again, these are just examples and you are free to change them; just be sure that there is some information for the user to see what is going on in the chatroom in addition to the chat messages.

Lastly the BotManager should also have a frame with a single button. Clicking on this button should call the method `killAllBots()`, a doomsday button if you will.
The bots themselves do not need a UI.

Do not spend more time than necessary on your UI! Do not make a fancy configurator; only a simple view is sufficient.

Observer pattern

It is no longer allowed to use Java's `Observer` class and `Observerable` interface as these are deprecated. Instead you should either use `PropertyChangeListener` or implement your own custom Observer pattern. Don't worry though, as changing to `PropertyChangeListener` is relatively easy as it is quite similar to `Observer` and `Observable`. More information about `PropertyChangeListener` and how to work with it can be found in the reader.

Requirements

In summary, here are the requirements:

- A main server.
- Chatrooms.
- A bot manager.
- Two kinds of bots:
  - Local bots
  - Migratory bots
- At least two kinds of behaviour for bots.
- A simple view for the chatrooms and the main server.

Now you might be wondering: what kind of number of chatrooms/bots should we be aiming for? During the demo we expect:

- At least 3 different chat rooms
- At least 15 different bots

This will show to us that your program properly works with a reasonable number of connections.

Important

It is very important that you properly document your code! We expect proper use of Javadoc and we also expect unit tests for any public methods/constructors.
We do not expect you to test trivial getters/setters nor is it necessary to test the view.

We **do** expect you to test the networking. A section about this called **Network Testing** will be added to the reader soon that contains all the information you should need for this.

We will be mainly looking at how you design your code and how you use Object-Oriented Programming to achieve your goals. It is way more important to properly design your program rather than having all kinds of fancy features. To make it faster

We will also be looking for any code smells throughout your code, so be sure to periodically check your code and refactor it when needed. A list of code smells can be found in the reader.

Extras

It is possible to earn 1 bonus point by adding extras. For this, you could think of adding human interaction (allow the user to type as well) or more sophisticated bot behaviours.
It is, of course, possible to earn a 10 without this bonus point.