# Project Steps

1. Group work (group size: 3): Requirement specification for ChatServer

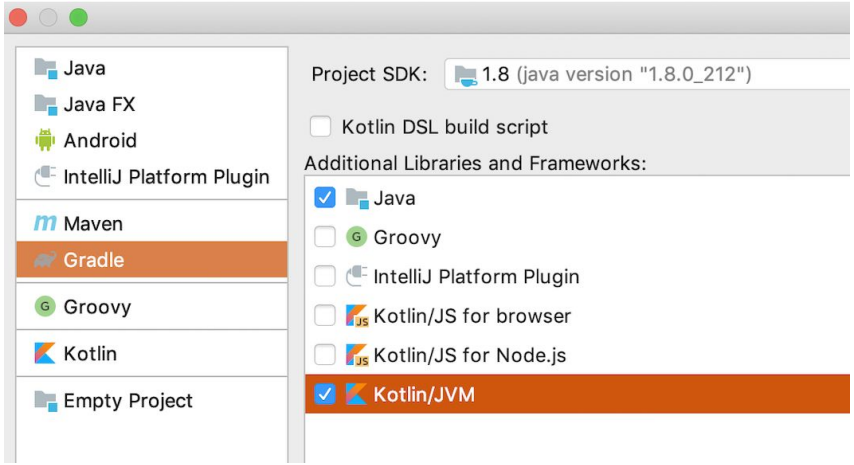
Write a requirement specification for the chat server. Find out the required functionality by interviewing Peter/Petri. When writing the specification, pay attention to possible exceptions. Divide your specification to sections that are easy to refer to (for example, “1.2 user commands”).

Return the specification to oma.

# 2. Create project for your ChatServer

Create Gradle project with Java and Kotlin/JVM checked. (We will be using json serialization from kotlinx -library and therefore will need the Gradle build system to take care of our depencies with external libraries)

Project name should be ChatServer + your initials like ChatServerPVe for Petri Vesikivi's chat server.



# Edit build.gradle file (add lines with yellow background) to enable use of kotlinx library

```
buildscript {
    ext.kotlin_version = '1.3.41'
    repositories { jcenter() }

    dependencies {
        classpath
"org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
        classpath
"org.jetbrains.kotlin:kotlin-serialization:$kotlin_version"
    }
}

plugins {
    id 'java'
    id 'org.jetbrains.kotlin.jvm' version '1.3.41'
}
……..
repositories {
    mavenCentral()
    jcenter()
}
```

```
dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
    testCompile group: 'junit', name: 'junit', version: '4.12'
    compile "org.jetbrains.kotlin:kotlin-stdlib:1.3.41"
    compile
"org.jetbrains.kotlinx:kotlinx-serialization-runtime:0.11.1"
}

compileKotlin {
    kotlinOptions.jvmTarget = "1.8"
}
compileTestKotlin {
    kotlinOptions.jvmTarget = "1.8"
}

apply plugin: 'kotlinx-serialization'
apply plugin: 'kotlin' // 'kotlin-android' for Android specific
```

# 3. Implement ChatConnector

Implement a ChatConnector that reads user input and creates an object of ChatMessage type. Your application should have a `main()` method where you create an instance of the ChatConnector and execute its run method.

```
fun main(args: Array<String>) {
    val chatConnector = ChatConnector(System.`in`, System.out)
    chatConnector.run()
}
```

# 4. Implement history using Kotlin object

Use Kotlin **object** to create `ChatHistory` singleton. Implement in that object public methods:

- `fun insert(message: ChatMessage)` : insert a new message (of class `ChatMessage` [that you need to create, too])
- `override fun toString(): String` : return the whole chat history as a nicely formatted string

`ChatMessage` objects need to have instance variables to support the features visible in the example run log. A constructor and `toString()` method are needed, think twice before introducing more methods. `ChatHistory` will need an instance variable of a suitable collection type to store all messages.

# 5. Implement `ChatServer` class

Implement `ChatServer` class that has a method `serve()`, which

- Listens to incoming connection requests using `accept()`
- Starts a new `ChatConnector` thread for each connection

Your `main()` method should look like this:

```
fun main(args Array<String>) {
    ChatServer().serve()
}
```

# 6. Make `ChatHistory` *observable* and `ChatConnector` *observer*

When new message arrives, it would be added to chatHistory. ChatHistory would call all of its observers. ChatConnector instances would output the message to PrintStream.

# 7. Implement Users

Implement singleton `Users` with Kotlin **object** that has methods for

- Inserting and removing username
- Checking if the username exists already
- `toString()` that returns the userlist as a nicely formatted string

(Use `HashSet` instance for storing the usernames internally in `Users` class.)

# 8. Implement support for json messages

Add @serializable to your ChatMessage -class. Parse all incoming messages into objects of ChatMessage type. Test your server with valid and invalid json messages. Connect multiple terminal sessions to see that messages gets spread to all chat participants.

# 9. Implement command interpretation to ChatConnector

Write business logic to chat connector to support the required command set. Consider also exceptions like wrong command etc.

# 10. Implement `ChatConsole`

Implement `ChatConsole`, which

- Registers as an observer to `ChatHistory`
- Prints out to `System.out` all chat messages in the conversation

# 11. Implement  TopChatter

Implement `TopChatter`, which

- Registers as an observer to `ChatHistory`
- Writes to console list of active users including the number of messages sent every time the list changes
- Challenge: modify the code so that it prints a list of four top chatters order by the number of messages they have sent