

Object Oriented Programming 2018

Stijn Mijland & Yannick Stoffers

Last build: 19th April 2018

Contents

I	Introduction & Setup	3
1	Organisation	4
II	Tools & Environment	5
2	Compilation	6
3	Version control	9
4	Continuous Integration	12
5	Integrated Development Environments	13
III	Exercises	15
6	Introduction	16
7	Text-based RPG	22
8	RPG + IO	30
9	Card Game	34
10	Final Assignment - Graph Editor	38
IV	Appendix	43
A	Git commands	44

I | Introduction & Setup

1 | Organisation

1.1 | Staff

Lectures will be provided by Prof. Dr. A. Lazovik and Dr. Rein Smedinga. Your practical work will be graded by the teaching assistants.

1.2 | Lectures

In the first two weeks, there will be two lectures per week. After, you will have 3 more lectures, one per week.

1.3 | Practicals

Every week you'll be working on a practical. Some labs span multiple weeks. In the week you start a new lab, you'll find a line in bold formatted as follows:

DL: YYYY-MM-DD, HH:MM

This is the time you have to have submitted that lab. You'll also find a line formatted like this:

DEMO: YYYY-MM-DD

On that day you'll have a 15-minute meeting with the teaching assistants. You will show them how your program works, and they ask you questions about what you've made. At the end of this session, you'll have your grade.

Demo's are typically the day after you've handed in your code.

1.4 | What to hand in

For all labs you hand in the code you've written. If the build server determines that your code runs (WARNING: You're responsible for verifying the build server can build your code), your **code submission** is complete.

The final assignment (Graph Editor) requires additional documentation. Create a structured document explaining what you've made, how you approached the project, the problems you encountered and the decisions you made.

1.5 | Tutorial Lectures

In the schedule you'll also find tutorial lectures. Tutorial lectures are lectures given by the teaching assistants about various important topics related to the course. The topics are:

GitHub Labs are handed in through GitHub. During this tutorial you'll be shown how a typical lab works, and you'll receive the basics of Java programming.

Graph Editor Design For the final assignment you'll be working on a fairly open assignment. How would your TA's do that assignment? Here you discuss the possible ways of doing the assignment, and the

Exam Preparation In the exam you'll also be asked questions about peculiar aspects of Object Oriented Programming. During this tutorial we'll be looking at those aspects, and we'll work through a practice exam.

II | Tools & Environment

2 | Compilation

Compiling Java files can be done with any Java compiler, usually `javac`. This is a command line tool that takes any amount of arguments and will compile all classes found in any of the provided input arguments. Each class will be compiled to its own class file. These class files are interpretable by the Java runtime environment (JRE). In case a class contains a main method, the JRE will be able to use that class as a starting point for the program.

During this course, however, we will not ask you to manually compile all your source files. Instead, we will be use Maven, a tool that allows for automatically compiling, testing, bundling, and more. In order to use this you only need to use a single command in order to build and test your program. A series of useful commands and their purpose is listed below.

In order to make Maven function you will need to create a POM file and maintain a proper directory structure. The latter is the easy requirement, as you will need to create have a `src` directory in the root directory of the project. This is the directory in which all source files should be located. Two subdirectories should exist: `main` and `test`, for the main project and tests respectively. Either should contain the folder `java`, inside which you can specify the Java source files according to normal package structure:

```
project-root-dir
+-- pom.xml
+-- src
    +-- main
    |   +-- java
    |       +-- package
    |           +-- App.java
    +-- test
        +-- java
            +-- package
                +-- AppTest.java
```

Command	Purpose
<code>mvn validate</code>	Validates the project for correctness and determines whether all required information is available.
<code>mvn compile</code>	Builds the project.
<code>mvn test</code>	Tests compiled sources. The sources should not be packaged.
<code>mvn package</code>	Bundles compiled sources into distributable format.
<code>mvn integration-test</code>	Process and deploy the package if necessary into an environment where integration tests can be run. (This is the command CircleCI uses.)
<code>mvn verify</code>	Run any checks to verify the package is valid and meets quality criteria.
<code>mvn install</code>	Installs the package into the local repository, for use as a local dependency for other projects.
<code>mvn deploy</code>	Copies the package to remote repository, for sharing with other developers and projects.
<code>mvn clean</code>	Removes the target directory. In other words, deletes local output of previous builds.
<code>mvn site</code>	Generates the site associated with the project.

2.1 | The POM file

As mentioned above, the POM file is an XML file named `pom.xml` and must be located in root folder of the project. The outer most tags must be `project`, of which the opening tag can

indicate some parameters such as schema version, and schema location.

Inside the `project` tag there are a couple of tags that every project should have. The `groupId`, specifying the organisation owning the product; the `artifactId`, specifying the name of the product; and the `version`. These three tags uniquely identify a project. Furthermore, properties of the build can be set, such as source encoding and Java Development Kit (JDK) version.

Additionally, plugins can be defined for the building phase. These plugins each have their own task and will modify the execution of any of the four phases, and may produce additional output. In the example below, we have specified that `maven-surefire-plugin` should be used. This plugin automatically tests the product using the specified tests in the `test` subdirectory of the source.

Finally, dependencies may be specified as well. This allows for portability of the product, while keeping the size of the source to a minimum. Before the building phase Maven will download and install all specified dependencies, such that the project can be successfully build. In the example below, `junit` is specified, as it provides a method for automatically testing source code.

Listing 2.1: Example POM file

```
1 <!-- POM specification -->
2 <project
3   xmlns="http://maven.apache.org/POM/4.0.0"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
6                       http://maven.apache.org/maven-v4_0_0.xsd">
7
8   <modelVersion> 4.0.0 </modelVersion>
9
10  <!-- Names -->
11  <groupId> nl.rug.oop.helloWorld </groupId>
12  <artifactId> helloWorld </artifactId>
13  <version> 1.0-SNAPSHOT </version>
14  <name> helloWorld </name>
15
16  <!-- implementation details -->
17  <packaging> jar </packaging>
18  <properties>
19    <project.build.sourceEncoding> UTF-8 </project.build.sourceEncoding>
20    <maven.compiler.source> 1.8 </maven.compiler.source>
21    <maven.compiler.target> 1.8 </maven.compiler.target>
22  </properties>
23
24  <build>
25    <pluginManagement>
26      <plugins>
27        <!-- Tests -->
28        <plugin>
29          <groupId> org.apache.maven.plugins </groupId>
30          <artifactId> maven-surefire-plugin </artifactId>
31          <version> 2.19.1 </version>
32        </plugin>
33      </plugins>
34    </pluginManagement>
35  </build>
36
37  <!-- Dependencies -->
38  <dependencies>
39    <!-- JUnit unit testing framework -->
40    <dependency>
41      <groupId> junit </groupId>
42      <artifactId> junit </artifactId>
43      <version> 4.12 </version>
44      <scope> test </scope>
45    </dependency>
46  </dependencies>
47 </project>
```


3 | Version control

Version control is something that allows for easy file version management. The advantage that it offers is that multiple developers can work on the same project, and even the same file, at the same time. For this course we will be using Git for version control, and Github for repository management.

Git keeps track of the status of all files in the working directory, and the status of all known connected repositories (see flow diagram below). Let's start with the latter. Each setup consists of a local repository and a remote one. The local repository is located on the machine you're developing on. The remote repository is somewhere else, and for this course located on Github's servers. The local repository can either be behind or ahead of the remote repository. In case it is behind, the remote repository contains changes to the project that have not been made locally. In other words, someone else contributed to the project, and you have not synchronised your local copy yet. When the local repository is ahead, you have made changes, but have not made these available to your collaborators. In other words, only you have the changes available to you. All changes are bundled in commits and synchronised by pushing (upload) or pulling (download) such commits.

Git keeps track of your local files. Each time you modify a file, git marks the files as *changed*. Once you are satisfied with all your changes, you can stage the files you wish to commit. Git marks those files as *staged* adds them to the so called *index*. The next step is to explicitly commit all staged files. Git requires each commit to be accompanied by a short description, stating what has changed in that commit. Note that these descriptions are meant to be descriptive, such as *'bug fix: broken document layout in reader'*. Don't describe each changed line, those changes can be requested by accessing the full details of a commit. After committing the staged files, those files will be marked as *unchanged*, and your local repository will be marked as *ahead* of the remote repository. As mentioned before, pushing to the remote will then upload all local commits not present on the remote, and thus synchronise all progress.

Git also allows a local repository to be both ahead and behind of the remote simultaneously. This is rather straightforward, as it means that your local repository contains changes not present on the remote and vice versa. In this case, you must first pull the changes from the remote to your local repository, before you are allowed to push to the remote. Git handles merging of all changes automatically, meaning that in most case are changes are merged without you having to copy-paste a single character.

Merging, however, can fail under specific circumstances. Namely, when two developers have changed the same line of the same file manually. Git won't know which change is preferred, so it will modify the file to contain both changes, highlighted and separated (see below); and mark the file as *changed*. You will then first have to solve the merge conflict, by removing the appropriate lines, and committing the file once more. Note that merge conflicts can only occur while pulling changes, which is also the reason you must pull before you push.

A file involved in a merge conflict, of which an example is shown below, has a certain structure. Both conflicting changes are separated and surrounded by some highlighting.

```

2 class Example {
    public static void main (String args) {
4 <<<<<< HEAD:mergetest
    // Committed by developer A
6     System.out.println ("This is but an example...");
    =====
8     // Committed by developer B
    System.out.println ("...of merge conflict.");
10 >>>>>> 4e2b407f501b68f8588aa645acafffa0224b9b78:mergetest
    }
12 }

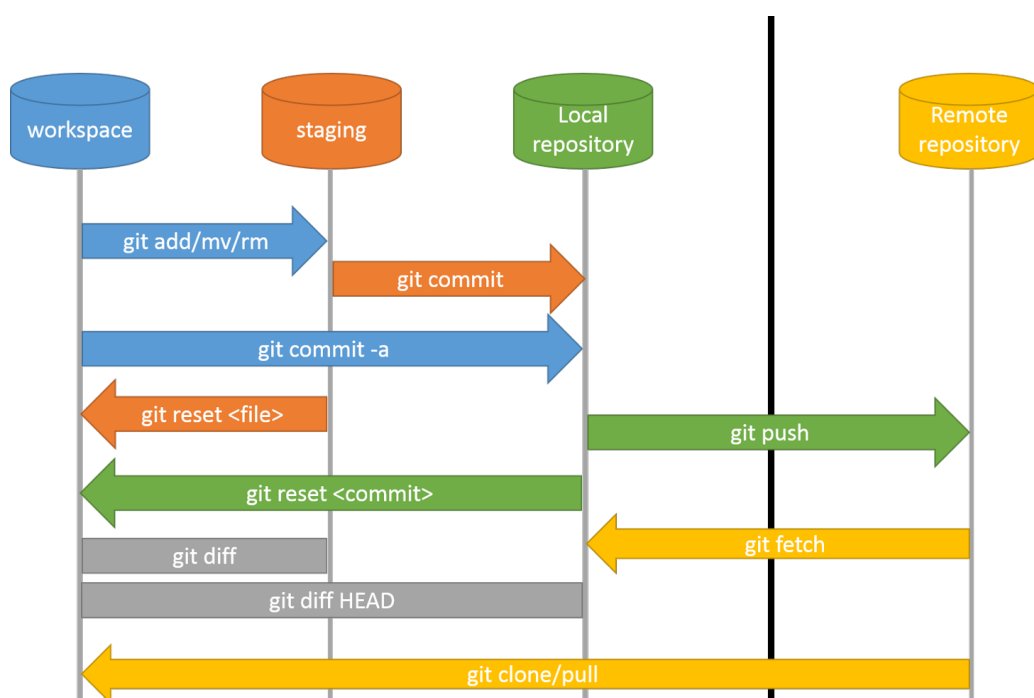
```

Below is a data flow diagram displayed of all git operations. It also contains some additional useful operations. Reset allows one to unstage a file (remove it from the index), or go back to a previous state by undoing a commit. This only undoes commits locally, and only temporarily. Reverting a commit is possible, but generally not needed. Diff allows one to compare the working directory with the index or with the head of the repository. In other words, it lists all changes done to a file. Fetch allows one to download all new commits from the remote, but not merge them immediately. Making it possible to first download changes before committing local changes, preventing merge conflicts. Note: a pull performs both a fetch and a merge.

3.1 | Branches

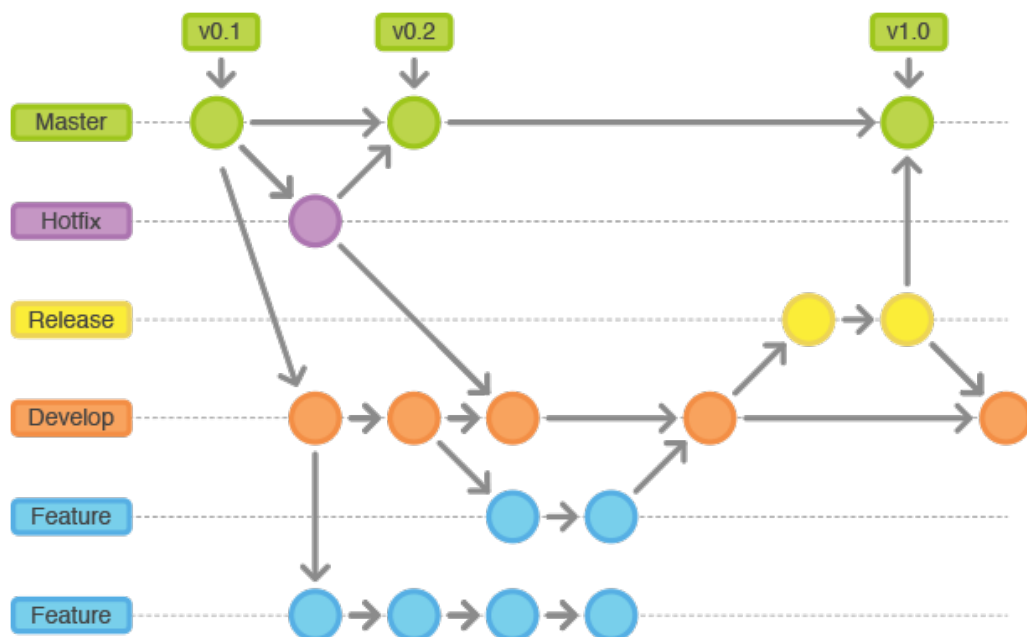
Branches in git are a manner of splitting off of the main stream of commits. Essentially copying all contents of the repository into a *new sub-repository*, grants it the possibility to develop multiple versions of the same program at the same time. It's main purpose is to let development teams work on different features on different branches. This allows for testing of a single feature at the time, without interference of other dysfunctional, unfinished features.

It is possible to do the same thing with public repositories of other developers, for which you do not have push permissions. In this case you actually copy the entire contents into your own repository, allowing you to continue someone else's work. This is called forking.



At any point a new branch can be created simply by checking out into a new branch. This will only create the branch locally, and contains everything the branch you're splitting off from. From that moment you will be working on the newly created branch, until you checkout into another branch. Both branches will coexist, but your commits will be added to the branch you're working on, not to any other branch.

Any two branches can be merged at any point. The merge command even allows for merging multiple branches at once. In case you have split off from a branch you don't have push permissions for, or have forked someone's repository, and you feel your additions are worthy of integrating in the original branch, you can make a pull request. After someone with push permissions for the branch into which the pull request is made, verifies your contributions, they will be able to merge from the branch you have been working one, relatively easily. This is how you are required to submit your projects.



4 | Continuous Integration

Continuous integration refers to a setup in which a server is connected to your repository. This server is notified every time a new commit is pushed to the repository. It then proceeds to pull the changes, and builds and tests the project. Finally, it notifies the repository server of the results. The main purpose of such systems, is that each commit gets build and tested at least once, without having to do this manually.

For this course, we have a CircleCI subscription authenticated for the entire Github organisation. This service will automatically build and test all your commits, such that you are at all points aware of the status of your project. Additionally, the service has to be able to successfully build each pull request before it can be merged.

5 | Integrated Development Environments

An integrated development environment (IDE) is an editor that is specialised software for developing in a specific language. Most modern IDE's have adopted support for multiple languages, although the core mechanics remain the same.

This means that they mostly provide auto-complete suggestions and documentation integration of standard libraries. Furthermore, they usually provide git integration as well. The purpose of IDE's is to allow for faster programming, by generating code and completing statements for you.

For Java the three most common IDE's are Eclipse, IntelliJ IDEA, and NetBeans¹. It is strongly recommended to choose either one for developing in Java, although Eclipse is the only one installed on university machines (both Windows and Ubuntu). Make sure to set your project settings to compile using maven, such that your IDE automatically does the directory management for you.

¹As established by a survey in Q2 2016.

III | Exercises

6 | Introduction

The hello world program is an important program for any programming language. The ability to create a program, compile it, run it, then see its output shows that your programming language works, which is nice in itself.

This exercise is not as simple as that. If you want to, you can find hello world in Java anywhere on the internet. This hello world is a hello to the world of Java (which is far larger than just printing) and the way you submit programs in this course.

6.1 | Before you begin

Before you begin any project, it's a good idea to set up the project first. This will save you time in the long run. Git projects like to be cloned in empty directories. If you've been entered into a group, and you've been given a repository, that means you can now clone the assignment.

In order to do so, start by cloning your own private repository. Type `git clone https://github.com/rug-oop/group_1_s0000001.git` in the command line interface of your choosing. The exact url you need, you can find on the GitHub website (on the page of your own repository there should be a *clone or download*-button, which will provide you with the exact url). Make sure to use the https url, or the ssh url if you've set up an ssh key. Enter your username and password, such that the local repository is created and automatically linked.

Next, make sure you navigate to the root directory of the newly created repository. Check out into the development branch using `git checkout development`. This will make sure you're tracking the correct branch.

Specify an additional remote repository as well: `git remote add assignments https://github.com/rug-oop/Assignments.git` (for the ssh url, please find the url on the GitHub website). Finally pull from this remote by using the command `git pull assignments master`. This will download all the changes in the assignments repository and places them into your development branch. There is however a slight chance that git will cowardly refuse to merge the downloaded changes with your own repository. In that case you can execute the command `git pull --allow-unrelated-histories assignments master`. That should solve the problem.

Now that you've cloned the assignment, you'll notice that your folder contains the following files: A folder called `introduction`, and a file called `circle.yml`. The folder contains a `pom.xml` file and the `src` folder.

In this course we use the domain `oop.rug.nl`. This exercise is called introduction. That means that your `pom.xml` is in the folder `introduction`, you should put your source files in the folder `introduction/src/main/java/nl/rug/oop/introduction`.

6.2 | A Main Method

Java programs are organised in classes. In order to start writing code, you first need to create a class. Create a file called `Main.java`. Now type:

```
public class Main {  
2 }  
}
```


When the java compiler finds the qualifiers `public` and `class` together, it expects the name of that class (`Main` in this case) to be in a file called `Main.java`. A `public class Card` would be in a file called `Card.java`, for example.

Inside the class `Main`, write (respecting indentation):

```
2 public static void main(String[] args) {
}
```

This is what the main method looks like in Java. A program starts when main is called. You could now:

1. Compile your class with `mvn compile`, generating `Main.class` inside the `target/classes` directory.
2. Run your program with `java -cp target/classes/ Main`.
3. ???
4. See nothing happen because your program is empty.

6.3 | Adding Print Statements

To turn your program that does nothing into a program that does something, we add print statements. You could go classic: `System.out.println("Hello World!");`. For this exercise, however, we want you to:

- Loop over the command line arguments.
- Print the reverse of each one

6.4 | Iterable

Many data structures in Java support the `Iterable`-interface. Arrays, Lists, Sets, and a lot of others. In Java, there is a special for loop you can use for `Iterable` data structures: the `for-each` loop:

```
2 for(String argument : args)
   System.out.println(argument);
```

Arrays are a peculiar kind of type in Java, so they're not listed, but all other types listed in the documentation of `Iterable` can be looped over using the `for-each` loop.

6.5 | Reversing Strings

We are sorry to report that Java does not have a built in way of reversing strings. You're going to have to implement it yourself. We are also sorry to report that `String` is not `Iterable`, so you're going to have to iterate over it yourself.

```
2 private static String reverseString(String arg) {
3     StringBuilder reverse = new StringBuilder();
4     for(int idx = arg.length(); idx > 0; idx--)
5         reverse.append(arg.charAt(idx - 1));
6     return reverse.toString();
}
```

If you want to create a method that works like a C function, give it the qualifier `static`. If you have a method you don't want (or need) other classes to see, give it the qualifier `private`. Return types and arguments work the same as in C. This function goes inside your class. That should give you something like this:

```
1 public class Main {
2     /**
3      * Return a new string that is the reverse of the argument arg.
4      */
5     private static String reverseString(String arg) {
6         StringBuilder reverse = new StringBuilder();
7         for(int idx = arg.length(); idx > 0; idx--)
8             reverse.append(arg.charAt(idx - 1)); //add characters in reverse order
9         return reverse.toString();
10    }
11
12    /**
13     * A simple program that prints its command line arguments in reverse
14     */
15    public static void main(String[] args) {
16        for(String argument : args)
17            System.out.println(reverseString(argument));
18    }
19 }
```

6.6 | New Requirements

The only constant in software development is changing requirements. The requirements now are:

- Loop over the command line arguments using the for each loop
- Print the reverse of each of them
- Read lines from input
- If you haven't encountered the line before print "You're so smart and intelligent!" and remember reading it.
- If you have encountered the line, print "Pffft, everyone knows that!"

You've been proposed the following approach:

```
1 public static void main(String[] args) {
2     for(String argument : args)
3         System.out.println(reverseString(argument));
4     Memory mem = new Memory();
5     Scanner scan = new Scanner(System.in);
6     while(scan.hasNextLine()) {
7         String line = scan.nextLine();
8         if(mem.canRemember(line))
9             System.out.println("Pffft, everyone knows that!");
10        else {
11            System.out.println("You're so smart and intelligent!");
12            mem.remember(line);
13        }
14    }
15 }
```

There are a few new things here. To get input we use the Scanner class, which contains various convenient ways of interpreting input. Right now we only use it for reading lines from input, but it has many other convenient methods.

If you were to replace your main function with this example, however, it would not compile. You'd receive an error like:

```

1 [ERROR] COMPILATION ERROR :
2 [INFO] -----
3 [ERROR] /Users/yannick/Documents/OOP2017/Demo/group_1_s0000001/introduction/src/main/
   java/nl/rug/oop/helloWorld/Main.java:[36,17] cannot find symbol
   symbol:   class Scanner
   location: class Main
5 [ERROR] /Users/yannick/Documents/OOP2017/Demo/group_1_s0000001/introduction/src/main/
   java/nl/rug/oop/helloWorld/Main.java:[36,36] cannot find symbol
   symbol:   class Scanner
   location: class Main
7 [INFO] 2 errors
9

```

To fix it, place the following statement at the top of your file: `import java.util.Scanner`. Java is organised in packages, and only the packages in `java.lang` are imported by default. All other packages (including ones you make yourself) need to be imported manually.

Besides the scanner and the import statement, we also have a type memory. This type can not be found in the Java libraries, so you'll have to make it yourself.

6.7 | An own type

With types programmers are able to create programs that are no longer about, say, collections of strings, but about people and products, for example. This helps everyone involved in understanding the program and its logic.

In this case, we would like you to make a type memory. What you know about memory is that it:

- Has a constructor `Memory()` with no arguments.
- Has a method `canRemember(String line)` which returns a boolean.
- Has a method `remember(String line)` which returns void.

Here's how that looks.

```

import java.util.HashSet;
2
public class Memory {
4
    private HashSet<String> memory;
6
    public Memory() {
8        memory = new HashSet<>();
    }
10
    public void remember(String said) {
12        if(!memory.contains(said))
            memory.add(said);
14    }
16
    public boolean canRemember(String said) {
        return memory.contains(said);
18    }
}

```

Again, like with any `public class` this should be placed in a file called `Memory.java`. It's also advisable to keep `Main.java` in the same folder if you want to avoid importing memory in `Main`.

6.8 | Comments and Compilation

Functionally speaking we might be done, but good code also has comments. Add some comments to explain what happened here. When you've done that, it's time to compile the project.

To do so, navigate to the folder that contains the `pom.xml` and type `mvn compile` in a terminal. If you've done everything correctly, there should be a line in the output saying `[INFO] BUILD SUCCESS` as one of the very last lines.

Now it's time for some tests. Run the command `mvn integration-test` in your command line interface, pay attention to the output and what changed with respect to the compile output.

If you've followed this guide so far, you should be able to run your program in either of the following ways:

- In the folder with the pom, calling `mvn exec:java -Dexec.mainClass="Main" -Dexec.args="test arguments"` (where `exec.args` are the command line arguments to the program).
- Navigating to `target/classes` and calling `java Main <test arguments>` in a terminal, where the arguments after `Main` are the command line arguments to the program.
- Invoking `java -cp target/classes/ Main <test arguments>` in a terminal.
- Pressing the run button in an IDE.

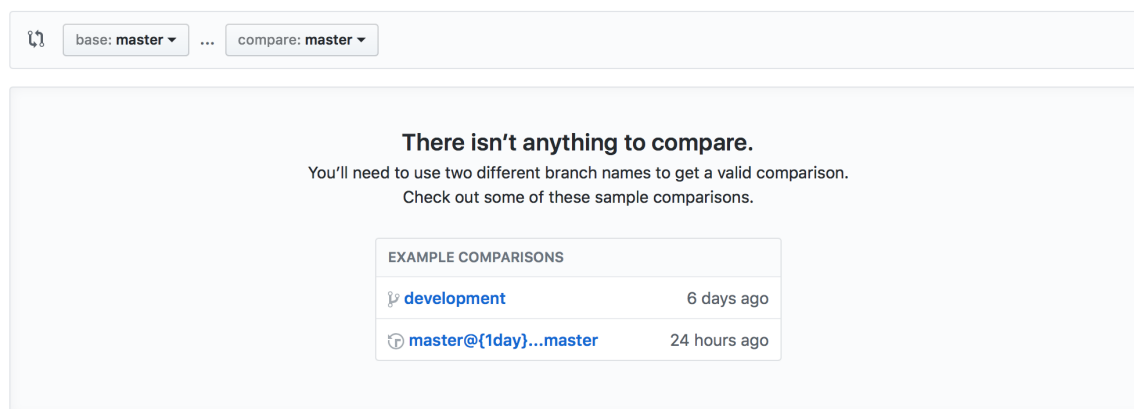
Try the program for a bit, and after confirming that it works flawlessly, submit a pull request.

6.9 | Handing in

You submit your program by creating a pull request on the GitHub website. Navigate to your repository and select the code tab. There should be *new pull request* button. Make sure that the base and the comparison branch are set properly. Incorrect:

Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).



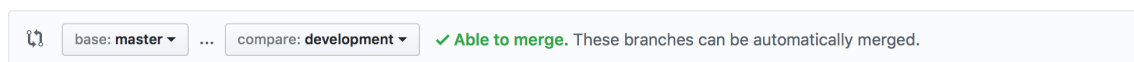
The screenshot shows the GitHub 'Comparing changes' interface. At the top, there are two dropdown menus: 'base: master' and 'compare: master'. Below these, a message states: 'There isn't anything to compare. You'll need to use two different branch names to get a valid comparison. Check out some of these sample comparisons.' Below this message is a table titled 'EXAMPLE COMPARISONS' with two rows:

EXAMPLE COMPARISONS	
development	6 days ago
master@{1day}...master	24 hours ago

Correct:

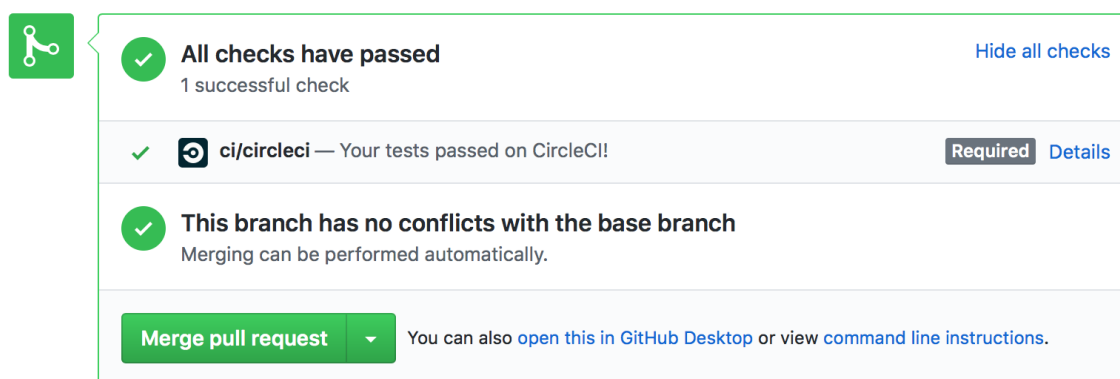
Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you can also [compare across forks](#).



The screenshot shows the GitHub 'Comparing changes' interface. At the top, there are two dropdown menus: 'base: master' and 'compare: development'. Below these, a green checkmark icon is followed by the text: 'Able to merge. These branches can be automatically merged.'

Finally create the pull request, and wait for CircleCI to build your latest commit, if that is still needed. The status on GitHub will automatically change. A successful submission should look similar to this:



The screenshot shows a GitHub pull request status. On the left is a green icon with a white branch symbol. To the right, there are three green checkmark icons in circles, each followed by a status message:

- All checks have passed**
1 successful check [Hide all checks](#)
- ci/circleci** — Your tests passed on CircleCI! [Required](#) [Details](#)
- This branch has no conflicts with the base branch**
Merging can be performed automatically.

At the bottom, there is a green button labeled 'Merge pull request' with a dropdown arrow, followed by the text: 'You can also [open this in GitHub Desktop](#) or view [command line instructions](#).'

7 | Text-based RPG

In this lab you will be making your own text-based RPG. This assignment is intended to get you to experiment with inheritance and polymorphism. The more you make use of interfaces and (abstract) super classes, the better.

In this RPG you'll be acting as the player. You'll make your way through all kinds of rooms, and you'll run into some of the most fantastic creatures known or unknown to man. If you manage to reach the final room, you win.

7.1 | Before we begin

7.1.1 | Setting up

This time, there is no need to clone any repository. If you've successfully completed the steps from the previous assignment, you should have a single repository with two remotes already. You can simply pull again from the assignments repository using `git pull assignments master`. This should set you up completely for this assignment.

Your working directory should now contain a new folder, namely *rpg*. This folder will once again contain a pom file and a src folder, you may notice that the `circle.yaml` file has been updated as well. For this assignment this is where you will write your project in. Find the empty file `rpg/src/main/java/nl/rug/oop/rpg/Main.java`. This file should contain the main class of your program.

7.1.2 | Incremental Maintenance

This exercise has been spaced so you can focus on a small subproblem each time. That way, it's easier to manage the changes. Therefore, try to maintain a pleasant working environment. After every step:

- Add comments to your code if you haven't done so yet.
- Integrate your changes into the main program so they're visible and usable.
- Verify that your code still compiles.
- Make sure your code still looks pretty.
- Commit your changes.
- Push the commit to your remote in order to synchronise your progress, and have a back-up of course.

7.2 | Rooms and You

Before we can start playing, we need something to walk around in. We will start with a single room. What is the property of a room? Rooms can be `inspected`. What that does is print a descriptive piece of text to `System.out`.

You've now created your room-class. Also create a Player class that keeps track of what Room it is in. Use these two classes (and potentially a Main class) to create a small program in which you can do the following:

```
1 What do you want to do?
  (0) Look around
3 0
  You see: A white room with a red door and a black door
5 What do you want to do?
  (0) Look around
```

Remember that our rooms should be somewhat reusable, so don't hard code the description in Room. Hard code it in Main, rather, so you can theoretically load it from file some time in the future. Doing so is better practice.

7.3 | Implementing Doors

Can you look around the room? Good! That means you can start adding doors. One of the things to know about doors is that they too can be `inspected`. That means you can reuse the code you wrote for Room. More than likely, you've given Room a `private String description`; which you print when `inspect` is called. We are going to generalise that property.

7.3.1 | Inspectable

Create a class `Inspectable`. Since an inspectable isn't a thing, make the class abstract.

```
abstract public class Inspectable {
```

Move the field `description` to this upper class, and move the method `inspect` as well. Also add a constructor to initialize `description` in `Inspectable`. Now you can change the line

```
1 public class Room {
```

into the line

```
1 public class Room extends Inspectable {
```

You can do the same for Door. Note: the program will not compile now, you'll need to finish the next section to make it compile again.

7.3.2 | Constructor Chaining

As you know, classes have constructors that are responsible for correctly initialising the fields in that class. However, if you extend a class, your class also provides the methods and fields of that other class, even though it is not responsible for their initialisation.

The piece of code that is responsible, is the constructor of the super class. Therefore, object-oriented convention is to first construct the super class before constructing the current class.

The way this order is enforced, is through the following rule: the first call in a constructor, must be a call to a different constructor. This can be a constructor in the class itself (for example, if you have a default argument you can pass it to the constructor with all arguments to reuse that constructor), but it can be¹ a constructor of the super class.

Note that even though the first call in a constructor must be another constructor, that a class like `Inspectable` won't have a call like `super()`. This is because Java infers that if there is no explicit constructor call, the empty constructor of the super class is sufficient to initialise the parent object.

Unfortunately it is not always possible to make objects with empty constructors. In that case, the inferred constructor is unavailable, which will cause a compilation error. For example: the constructor of `Inspectable` needs a `String`. That means that the code as it is right now will not compile (try `mvn compile` and inspect the error message). To fix this, change the constructors of `Room` and `Door` to contain the line (provided description is an argument) `super(description)`; If we leave out the comments, `Door` now probably looks like this:

```
public class Door extends Inspectable {  
    public Door(String description) {  
        super(description);  
    }  
}
```

7.3.3 | Adding Doors

With doors implemented, they can be added to the room. Add a `List<Door>` as a field to the room. Make sure to assign a sensible implementation in the Constructor (`ArrayList` is almost always a good choice for a `List`). Also create a method to add doors to the room. Then, create a method to inspect doors.

Augment the gameplay options with another option. The program should now work something like this:

```
1 What do you want to do?  
  (0) Look around  
3  (1) Look for a way out  
  1  
5 You look around for doors. You see:  
  (0) The red door  
7  (1) A black door  
What do you want to do?  
9  (0) Look around  
  (1) Look for a way out
```

Take care that when you loop over your list of doors, that calling `inspect` on `null` will cause a `NullPointerException`. Try adding `null` as a door and see what that looks like, and then build in checks to prevent such disasters from happening.

¹And unless you want a stack overflow due to a constructor cycle it eventually must be.

7.4 | Making Doors Work

Now that we can inspect doors safely, we can also enter them safely. If a `Player` `interacts` with a `Door`, they go through it, placing them in another room. Give the `Door` a field for the room behind it, and make sure that when the `interact` method is called, the player moves to that room. The user interface should now look something like this:

```

What do you want to do?
2  (0) Look around
   (1) Look for a way out
4  0
You see: A white room with a red door and a black door
6  What do you want to do?
   (0) Look around
   (1) Look for a way out
8  1
10 You look around for doors. You see:
   (0) The red door
12  (1) A black door
Which door do you take? (-1 : stay here)
14 1
You go through the door
16 What do you want to do?
   (0) Look around
18  (1) Look for a way out
   0
20 You see: A dark room with dark doors

```

We advise you to implement `interact` by passing the `Player` as an argument. That way it will be much easier to interact back with the player, and there is no need to make the player a field everywhere, or to make it a global variable, which is even worse.

Don't forget to create separate methods for each of these choices. That way it will be easier to find where a certain action happens. Also pay attention that encapsulation is not violated, as it will be harder to fix it later.

7.5 | Interfaces

Besides classes, you can also use interfaces to inherit from. Interfaces are used to declare kinds of methods a class can use. A good way to use an Interface in the preceding program is with `Interactable` for the method `interact`.

A common example of an interface is the `Comparable` Interface which indicates that a type is comparable with another type (usually itself). This one is particularly useful as it enables many ordering based algorithms and data structures.

If you find yourself repeating the same name of a method often, it's not a bad idea to create an interface for it. This helps people understand what your type does. The compiler will verify that classes that claim to implement an interface do indeed have an implementation.

Put the method `interact(Player player)` in an interface called `Interactable` and implement the interface in `Door`. You will use this interface when we go to NPCs.

Listing 7.1: Occupiable.java - An Interface Example

```
1 public interface Occupiable {  
2     public void occupy();  
3     public void leave();  
4     public boolean isOccupied();  
5 }
```

Listing 7.2: Toilet.java - an Interface implementation

```
1 public class Toilet implements Occupiable {  
2     private boolean occupied = false;  
3     @Override  
4     public void occupy() {  
5         occupied = true;  
6     }  
7     @Override  
8     public void leave() {  
9         occupied = false;  
10    }  
11    @Override  
12    public boolean isOccupied() {  
13        return occupied;  
14    }  
15 }
```

7.6 | NPCs

If you want to add NPCs, repeat the steps you took with the doors: create fields in the Room, make the NPCs `inspectable` first, and then add `interactions` by declaring that NPC implements `Interactable`.

Remember to test your program between each step. Everything should have been smooth thusfar because you went slowly. With everything you code you have a small chance of introducing a bug. If you have a lot of bugs, they start to affect each other, and the number of problems expands more quickly than the number of features. Take it slow and you'll be done quickly.

Although NPCs should ideally be abstract, for now it's better to use an example implementation that just makes a print statement to verify they are working. This could give you a user interface like this:

```
1 What do you want to do?  
   (0) Look around  
3  (1) Look for a way out  
   (2) Look for company  
5 2  
   You look if there's someone here. You see:  
7  (0) A danish man with lush blonde hair  
   (1) Human head, body of a lion - yup, a sphinx  
9 Interact ? (-1 : do nothing)  
  1  
11 The creature is asleep so you can't interact with it.
```

```

What do you want to do?
13  (0) Look around
    (1) Look for a way out
15  (2) Look for company

```

Make sure you're happy with this stage, also in terms of code reuse, then continue.

7.7 | Multiple NPCs

Now make NPC abstract. Create two classes that extend from NPC. Make sure they do something different when you **interact** with them or **inspect** them. Add them both to the room, and see what happens. Note that since NPC is abstract **interact** from **Interactable** no longer needs an implementation.

Now try to come up with your own types. The only requirement we have is that they extend from NPC. You could use an abstract super class, which implements part of a behaviour. For example, things that can be on or off, such as switches or light bulbs. They could have a common super type that has a method **toggle**, which changes a boolean. The switch and the bulb could then change their behaviour depending on that boolean.

Don't forget that you can add methods to player which can be called by an NPC **interacting** with the player. That way you can add a health system, or a money system, or status effects. Just be sure that you have time for the other parts of the assignment.

We ask that you have at least 2 meaningful super classes, and preferably two interfaces as well. Since code reuse is only achieved if multiple classes extend a super class, does it make sense to have only one child of a class? Perhaps it does, as the presence of such a class allows for extending the program more easily later on. Try to be concious in these decisions.

7.7.1 | Callbacks

Up till now we've been interacting with NPCs and Doors in a room. We knew they were doors and npc's so there was no problem interacting with them. Suppose however, that we wanted an extra kind of functionality. For example, a shop. In this case we would need to add an entire extra list to each room and individually add all NPCs (or Doors) that sell items to that list also. While sometimes that is indeed what needs to happen, an alternative approach exists: call a method in player.

When player calls **interact** on an NPC, that is forwarded to a real NPC. Let's call that NPC a **ShadyVendor**. If that **ShadyVendor** implements **Shop**, we are allowed to call that **ShadyVendor** a **Shop**. More importantly, the keyword **this** in **ShadyVendor** can be passed to methods that require a **Shop** as an argument.

After Player called **interact** on the NPC, which turned out to be a **ShadyVendor**, the **ShadyVendor** now calls the method **offer(Shop shop)** in **Player**, on the **Player** passed to **interact**. The **Shop** it passes as an argument, is itself. This means it makes a call back on the argument of a method. This is what a callback refers to.

What this accomplishes is that **Player** can now on its turn perform a callback on the provided **Shop** to trade items, using the methods declared in **Shop**. It also safely converts an NPC into a **Shop**. Naturally, this can be done with any interface. The following example illustrates a callback, and will print "Help me!", not "What's happening?";

Listing 7.3: Example of a Callback - Callback Receiver

```
1 public class Person {
2     public void give(Animal animal) {
3         animal.pet(this); //an aggressive animal will attack
4     }
5     public void attack(Animal animal) {
6         System.out.println("What's happening?");
7     }
8     public void attack(AggressiveAnimal animal) {
9         System.out.println("Help me!");
10    }
11    public static void main(String[] args) {
12        new Person().give(new AggressiveAnimal());
13    }
14 }
```

Listing 7.4: Example of a Callback - Abstraction

```
1 public interface Animal {
2     public void pet(Person petting);
3 }
```

Listing 7.5: Example of a Callback - Callback Sender

```
1 public class AggressiveAnimal implements Animal {
2     @Override
3     public void pet(Person petting) {
4         petting.attack(this);
5     }
6 }
```

7.7.2 | Template Method

Another way of using inheritance for different types is using a template method. If you have a class with a couple of abstract methods which are called as part of an algorithm, that algorithm can be implemented in a super class, and the inheriting classes only need to implement the abstract methods:

Listing 7.6: Example of a Template Method abstract class

```
1 abstract public class Performer {
2     public void perform() {
3         sing();
4         dance();
5         jump();
6     }
7     abstract public void sing();
8     abstract public void dance();
9     abstract public void jump();
10 }
```

In Java 8 you can achieve the same using an interface. This is done by the new default methods, as shown below. In this example that makes more sense as the class Performer has no state.

Listing 7.7: Example of a Template Method interface

```

1 public interface Performer {
2     default public void perform() {
3         sing();
4         dance();
5         jump();
6     }
7     public void sing();
8     public void dance();
9     public void jump();
10 }

```

7.8 | Multiple Doors

You can do the same with doors as you can do with NPCs. A door could whisper at you when you go through it, or do damage to you, for example. Try to create a couple of different doors as well.

7.9 | Concise requirements

Below you find a summary of the functionality that we would like to see for the first deadline:

- Command-line interfacing.
- Inspectable rooms.
- Inspecting doors in rooms, and an option to go through one.
- Inspecting NPCs in rooms, and an option to interact with one.
- Multiple kinds of doors with distinct behaviour.
- Multiple kinds of NPCs with distinct behaviour.
- Use of inheritance for different kinds of NPCs and Doors.
- At least 2 abstract superclasses with different functionality.
- At least 2 distinct interfaces.

You are welcome to add your own twist to the RPG, create a real story or make the experience more enjoyable in whatever way you wish. Just make sure that at the demo you can present the above.

7.10 | Handing in

Create a pull request from compare: development into base: master and confirm that CircleCI accepts your pull request.

8 | RPG + IO

In this lab you will be expanding your RPG with extra functionality: Saving and Loading. You will also have the chance to improve or fix your RPG based on the feedback you received on the previous assignment. Perhaps this addition isn't necessary, but we'll be using Java IO. Try to maintain the good coding practices from RPG.

8.1 | Fixing RPG

Before you start on this exercise, it's advisable to fix the exercise it builds on. The usual metaphor is that of a foundation: build on a stable foundation and your building will last forever, but change the foundation with a building on top, or use a bad foundation, and your building will surely collapse.

8.2 | About Saving

A common procedure in many programs is writing things to file. Abstractly speaking, you're converting program data into a structured form, so you can convert it back to program data later. Converting program data to structured data is referred to as serialization. The reverse is called de-serialization. Many programming languages support this natively, among which Java.

In this lab serialization is used to make save files. By storing all game objects to file, and reinstating them later, the same game can be played multiple times, potentially for multiple endings or different experiences.

8.3 | Serialization In Java

Java has the interface `Serializable`. If you declare that your class implements this property, it can be written by means of an `ObjectOutputStream`. No methods need to be implemented - Java does this automatically. You do need to add the field

```
private static final long serialVersionUID = 42L;
```

This field is stored when the object is serialized to verify when loading that the class it was saved as is identical to the one it's loaded as. It's therefore important that the ID is unique for different versions.

When a Class declares that it implements `Serializable`, it can't be serialized unless all its fields are `Serializable`. If this is not the case, your program will throw a `NotSerializableException`:

```
1 java.io.NotSerializableException: java.util.Scanner
```

If there is a field that should not be serialized, place the keyword `transient` in front of it. Classes that represent local resources, such as IO streams and `Scanner` are not serialized because this format is also used in network communication, so those objects would not be valid if deserialized on a different machine.

Unfortunately the Java serialization format is intended for readability by Java programs, which means that it's difficult to read as a human, including when using a standard text-editor. This makes it harder to verify if your serialization succeeded. You will need to load the file to be sure.

Take a close look at the See Also-section of `Serializable`. Those sections should provide enough information about Serialization. In the event of an unexpected exception, there is a very high chance you're not the first one to experience it.

8.4 | Implementing Saving and Loading

Because Java has a good mechanism for serialization and de-serialization we will not re-implement it. Make every object you need to save `Serializable`. Then, expand the menu as follows:

```
2  What do you want to do?
   (0) Look around
   (1) Look for a way out
4  (2) Look for company
   (3) QuickSave
6  (4) QuickLoad
```

Implement `QuickSave` by writing the state of the game to a file. To that end, use an `ObjectOutputStream`. Use `.ser` as the file extension. Naturally, the `QuickLoad` will load the save file you save when calling `quicksave`. Be sure to give appropriate feedback to the user if something went wrong. See also `ObjectInputStream`, `FileInputStream` and `FileOutputStream`.

8.5 | More luxurious saving

Many modern games support named save files. Since your game was made just last week, so will yours. Naturally, we don't want to clutter our current folder with save files, so we need to store our files in a folder `Savegames`. Make sure your save files end up in the `Savegames`-folder, even if that folder does not exist (if that folder can't be created then you can't save of course).

Now that save files end up in the right directory, prompt the user to give in a file name upon choosing the save-function. Make sure to give appropriate feedback to the user about the file name. An example of the extended user- interface has been provided.

As you can see, when the user uses the luxurious load-function, they can load a `quickSave`, but also the named save made earlier. Make sure that only save files (with the `.ser` file extension) in the `Savefiles`-directory are available to load. See also the `list` method in `File`.

```
What do you want to do?
2  (0) Look around
   (1) Look for a way out
4  (2) Look for company
   (3) QuickSave
6  (4) QuickLoad
   (5) Save
8  (6) Load
5
10 Filename?
   room1
12 What do you want to do?
   (-1) Give up
14  (0) Look around
   (1) Look for a way out
16  (2) Look for company
   (3) QuickSave
18  (4) QuickLoad
   (5) Save
20  (6) Load
3
22 What do you want to do?
   (-1) Give up
24  (0) Look around
   (1) Look for a way out
26  (2) Look for company
   (3) QuickSave
28  (4) QuickLoad
   (5) Save
30  (6) Load
6
32 Which file? (-1 : none)
   (0) room1.ser
34  (1) quickSave.ser
```

8.6 | Functionality Warning

It might be the case that some of your functionality breaks when you load a save file. Be sure to verify that all program features work even after you've loaded a game.

8.7 | Concise Requirements

If there is anything you would like to add, feel free to do so. You're only required to do the following:

- Fixing last week's RPG (RPG's requirements also apply to RPG + IO)
- Make all necessary types Serializable

- Implement QuickSave
- Implement QuickLoad
- Save files in Savefiles-directory
- Named save files
- Showing only `.ser`-files available for loading
- Loading named files

8.8 | Handing in

Before you hand in, **remember that an empty try/catch statement is subject to severe punishment** and that throwing an exception where catching it would have been sensible is frowned upon.

Create a pull request from compare: development into base: master and confirm that CircleCI accepts your pull request.

9 | Card Game

This lab is intended to help you get started with making GUI's in Java. During the two weeks of this lab you will be making your own card game program based on the model-view-controller pattern. Half way through you will have a feedback session with your TA's.

9.1 | General Assignment Structure

Because you're allowed to pick any card game you want, and because card games often differ greatly in their features, we can't give you a completely detailed description of how to do this assignment. What we can tell you is that the most problems will arise when you work on the GUI. Therefore, we suggest you follow the following structure.

- Model the flow of cards in your game.
- Create a `PrintView` for your model so you can verify that it updates correctly. Use the Observer pattern (see also `Observer`).
- For each function, create an `Action` that makes that change in the model. Create a `JButton` for that `Action` , and add put the `JButton` in the user interface.
- **Verify that your application works as intended.**
- Create and view a `GamePanel` that extends `JPanel` that draws your game by adding it to a `JFrame` .
- **Verify that your views work as intended**
- If there is still time left, add extras.

This structure will allow you to check if your model works as intended before you put a lot of time into making the view, allowing you to focus on figuring out the view, rather than spending time on your model yet again.

9.2 | Project Management

The most important thing in any project is communication. If you're working together, agree on how you divide the work, and if you can't manage your task, be clear about it so you can look at it together. Apart from that, there are many small things that you can improve.

At some point you might need to evaluate how you organize your software projects. One of the ways you could streamline the process is by keeping track of what still needs to be done. This can be done by means of a backlog.

9.2.1 | Backlog

A backlog is a list of things you still need to do. The items in a backlog are ordered by priority. You can also add certain metadata such as the amount of work you expect a task to be, or the person that will complete the task. A backlog could look as follows:

Issue	Assigned
Creating the Introduction exercise	Stijn
Creating the RPG exercise	
Preparing the Assignments directory for Introduction	Yannick

Backlogs help you to keep an overview of the things you still need to add to your program. You can put in every small detail (i.e. names are drawn over each other), but that does run you the risk of crowding the backlog with too many small things, which makes you lose the overview. That does not mean having small issues is a bad thing, it just means that you need to manage your backlog.

Traditionally backlogs are created in Excel, but the networkedness of Google Sheets is probably more convenient, and if a spreadsheet is inconvenient for you, you can always try specialised tools such as Trello.

9.2.2 | Working with Backlogs

At the start of a project it's usually good to brainstorm a little about the things you want in your project, and put those in the backlog. When you're out of ideas, you arrange how you're going to work on the items in the backlog. After a while, you get back together and see how you got on, update the backlog, and assign issues again. If you run into something while working you can always add it to the backlog. When an issue is completed, it's usually good to keep it as a record somewhere to see what you've already done, even though you remove it from the backlog.

9.3 | Demo Program

If you want to make your program really nice, chances are you'll be spending a lot of time in the Java Swing Javadoc. This is not really recommended. For that reason, we've provided a demo program that should have all the drawing techniques you need for this lab assignment.

To make it easier to create packages with this program, **The directory structure has been changed** from the usual `nl.rug.oop.cardGame` to the simpler `cardGame`. A package model would be found in `cardGame.model`.

9.3.1 | Program Documentation

The demo program, `draw`, is a simple program: You can draw cards, which end up on the discard pile. You can shuffle the cards on the discard pile back into the deck. You can use buttons, or drag cards with the mouse. The buttons each have a hotkey. The code for `draw`, has slightly more classes than this functionality may seem to warrant. This is because `draw` is designed according to model view controller.

The model aspect of `draw` is a deck, a discard pile and cards. These are abstracted to a 'game' - the class `Draw`. Since `Draw` changes, `Draw` is observable. The `DrawPanel`, a `JPanel` is a graphical representation of this `Draw`, which contains all the logic for arranging the UI elements according to the size of the panel (which is a lot of work, consider it an extra).

To translate the simple `Draw` into a GUI, each card in that model is changed to a `PositionedCard`. These `PositionedCards` are managed in the `PositionModel`, which decorates the `Draw`-class. The `DrawPanel` draws these cards. On top of this `PositionModel` there is a `MovementModel`, which changes the location of the top `PositionedCard` in the `PositionModel`.

You are allowed to reuse and modify any part of the game you want to. For more documentation on the program, generate the javadoc for the program by calling `mvn javadoc:javadoc` in the `cardGameDemo` folder and opening `cardGameDemo/target/site/apidocs/index.html`

9.4 | About the Architecture

As mentioned the exercise is intended to work with GUI's. That means that you need to design the application according to the Model-View-Controller pattern. There are a few key points to stay mindful of when using MVC.

The model is a static representation of the game state. Nothing changes in the model without the influence of a controller.

The view represents the model. However, you may find that your model is missing data you need for drawing. If that happens, it's okay to have a draw model: an intermediate layer that expands on game elements with positions and sizes.

The view you display to the user can be based on a draw model. In that case, the Model-View structure is repeated, but this time at a more concrete level. The first is from game model to draw model, the second from draw model to view.

A Controller can look at the view to determine when to perform an action, but only changes the model. This is particularly common when mouse interaction is involved.

9.5 | About Custom UIs

The provided program only uses one `JPanel` for drawing. This has a reason. In Swing, every UI element is a subclass of `JComponent`. Swing uses `Layout Managers` to order these `JComponents`. Each Swing Component is important for the program, so they should not overlap. Swing is not a general drawing mechanism.

If you want to have a UI where things can overlap, such as the `BufferedImages` we get from loading the images of the cards, you need to define your own drawing function and to arrange these elements yourself. See also: `Painting in AWT and Swing` (you can ignore the AWT part), `The Java Swing Tutorials` (for tutorials about most Swing UI elements)

9.6 | Concise Requirements

The important elements are:

- Graphical User Interface. Does not need to be complicated. A simple menu is sufficient.
- Correct Model-View-Controller
 - Observer Model
 - Model changed through Controller only
 - View changed via Observer pattern
- Correctly Working Card Game

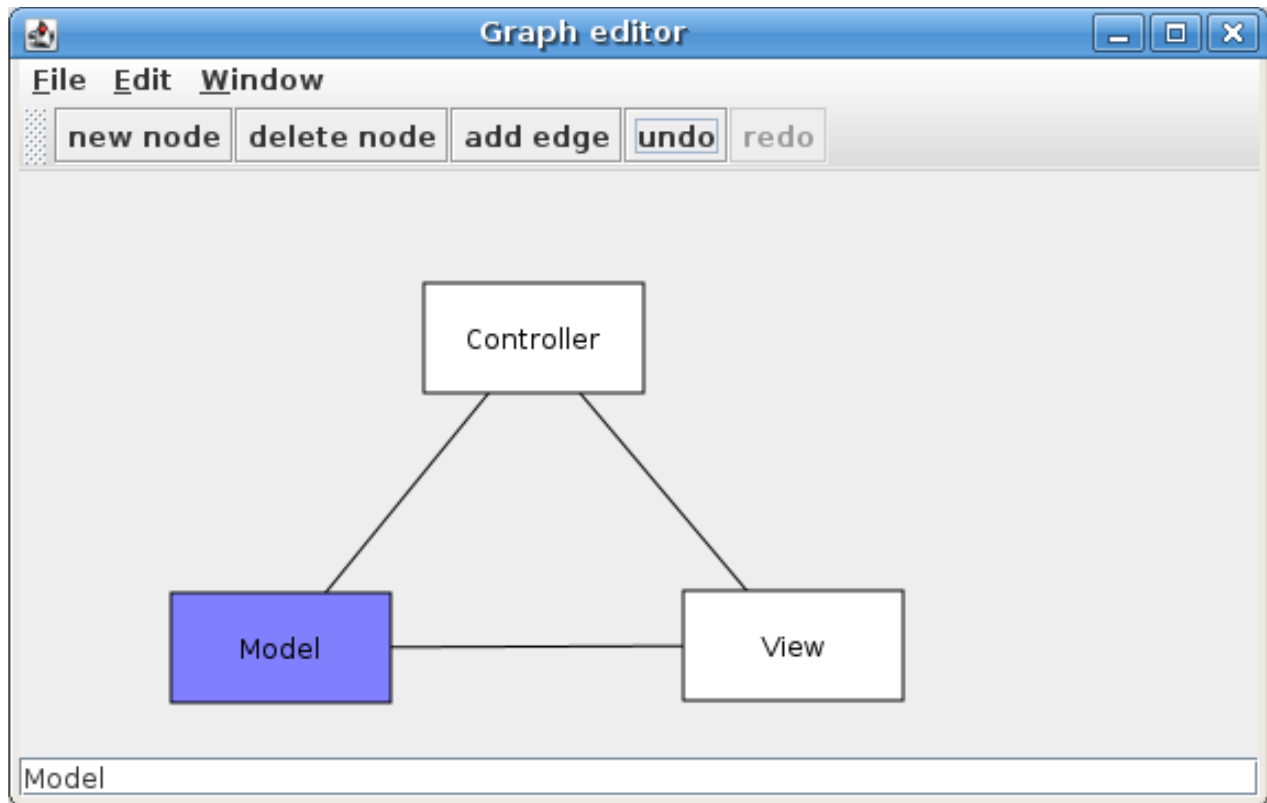
There are quite a few opportunities for extras here, even in terms of the normal requirements. Therefore, be sure to show them off in your demo.

9.7 | Deadlines

For the first deadline you need to hand in a graphical representation of your game. The important thing is that things change visibly. If partway through things break accidentally, that is allowed. The important thing is that you get familiar with drawing in Swing.

The same procedure applies for the second deadline, but there the entire assignment has to be completed. **Do not use Threads.** Threads are treated in other courses, and are quite hard to debug.

10 | Final Assignment - Graph Editor



For this assignment you will be creating a simple graph editor with a graphical user interface. This final assignment should test most of the skills you have learned in the past few weeks, such as Swing and MVC, Java IO, and the ability to complete a reasonably complicated project.

The graph editor assignment is split into two parts. In the first part you will be creating a graph model with saving and loading. The second part will include creating a graphical interface and user interaction.

10.1 | Graphs

The model part of this exercise is represented by an undirected, simple, unweighted graph without self-loops. You will be implementing it here.

1. Start with creating an empty class `GraphModel` which is the class to represent a graph. Also, create the classes `GraphVertex` and `GraphEdge` for the vertices and edges. Use a class `GraphEditor` for the `main`-method to start your application.
2. Add a list for all vertices and a list for all edges to `GraphModel`.
3. A vertex has a name and an edge always connects two vertices. Add variables, constructors, getters and setters to `GraphVertex` and `GraphEdge` for this
4. A vertex will be represented by a rectangle that has a size and location. to do this it is practical to use the `java.awt.Rectangle`. Make sure that every new `GraphVertex` object has a default name, size, and location.

5. Add methods to the `GraphModel` for adding and removing vertices and edges. Remember that for removing a vertex also every connected edge has to be removed.

10.2 | Saving and Loading

Now that we have a model, we will take a look at saving and loading. There is an example on Nestor that uses the following format:

- A line with two number: the amount of vertices and edges.
- For every vertex: A line with four numbers followed by the name of the vertex. The numbers are the coordinages, the height and the width of the vertex.
- For every edge: a line with the indices for the two vertices that the edge is connected to.

Proceed as follows:

6. Create a method `save` in `GraphModel` that saves a graph to file.
7. Create a method `load` in `GraphModel` that opens a graph from file. Also add a constructor that uses this method to immediately load a graph from file.
8. When the program is executed by using `java graphEdit filename` the graph from the file `filename` has to be loaded and used.

10.3 | The interface

Now we will process with the beginning of the `GraphEditor` that we made in the previous part of this assignment. Since we already have a working model and saving and loading functions, we can now start working on the interface part of the `GraphEditor`.

9. Start with creating a class `GraphFrame` that inherits from `JFrame`. This will be the main frame of the editor. Make sure that the frame has an appropriate title, a proper default size and a menu.
10. Create a class `GraphPanel` that inherits from `JPanel`. This is where all the vertices and edges will be drawn.
11. Make sure that the `GraphPanel` is visible in the main frame.

In a typical editor, it is possible to perform certain actions such as open, undo or add a vertex. An action in Java is represented by an object that inherits from `AbstractAction`. See also `Action`.

12. Create in the constructor of `GraphFrame` the action objects for adding vertices and make these actions available by using a menu and/or toolbar.

Now lets make sure that the graph will be painted in the `GraphPanel`. We will do this by implementing the method `paintComponent(Graphics g)`. This is the method that will draw the graph and it will be called whenever the method `repaint` is called. The method `repaint` is called whenever the frame is shown. You can also call this method yourself which is, for example, useful whenever the graph has changed.

13. It is, of course, important for the `GraphPanel` to know what graph has to be shown. Make sure that `GraphPanel` knows which `GraphModel` it is showing. Also add a method `setModel` to `GraphPanel` to be able to change which `GraphModel` the panel is showing.

14. When something in the graph changes, this also has to be shown in the graphical interface. Make sure that when something is changed in the graph, the method `repaint` is called. You should do this by making the `GraphModel` `Observer` and the `GraphPanel` `Observer`. Keep in mind that whenever another graph is shown, the `GraphPanel` should become an `Observer` of the new `GraphModel` object.
15. Make sure that the application remembers if there is a vertex selected and which vertex it is. The best way to do this is by making a new class called `SelectionController`. Don't think too much about the implementation of this class yet, for now it is enough to just have the class.
16. Create a method that can draw all the vertices of the graph. Make sure that this method will be called whenever the method `paintComponent(Graphics g)` is called. Your model already knows all the information of every vertex so the only thing that has to be done here is retrieving this information and using this to draw the vertices. Drawing is done by using the `Graphics` object. A vertex is drawn by first drawing a rectangle and then the name of the vertex. Also make sure that a vertex is drawn in a different color when it is marked as the selected vertex. For more information about drawing, read `Painting in AWT` and `Swing`.
17. Also create a method that draws the edges of the graph. Keep in mind that, like in real life, everything that is drawn first ends up behind something that is drawn later.
18. Now we will implement `SelectionController` to be able to select vertices. To do this `SelectionController` has to implement the interface `MouseListener`. The `GraphModel` knows all the positions of all the vertices so by using this information you can know if you clicked on a vertex and if so, which vertex it is. When you have clicked on a vertex, you can set this one as the selected vertex and repaint the view.
19. It should also be possible to drag vertices. This is also done by using a `MouseListener`. Keep in mind that dragging a vertex means that the information of that vertex in the model has to be updated.
20. Now that we are able to select vertices, we can also add actions for removing a vertex and adding an edge. Make sure that the buttons for this are only enabled when a vertex is selected. When the user wants to add an edge, a line should appear from the selected vertex to the cursor. The edge will only be made when another vertex is selected.
21. Think of a way to change the name of a vertex.
22. Think of a way to remove an edge.
23. Add an action that creates a new frame with exactly the same model. Changes made in one model should also appear in all the other frames.

10.4 | Undo and Redo

All modern editors have an undo and redo functionality. When you are already using a Model-View-Controller design architecture, it is relatively easy to add this functionality. The idea is to create for every operation a class that represents the operation. The classes should inherit from `AbstractUndoableEdit`. The constructor of the class takes care of the operation, while the methods `undo` and `redo` make sure that the operation is either undone or redone. These operations can then be added to an `UndoManager`.

24. Add an `UndoManager` to `GraphModel`.

25. Every operation in the model should use operation classes.
26. Make sure that Undo and Redo can be used from the menu.

10.5 | Extra

Finally you can add some extra features. These are not mandatory, but it can give you some bonus points. Think of things like:

- Keyboard shortcuts like [Ctrl]+[n], [Ctrl]+[z], etc.
- Copy and paste vertices.
- Try to make the interface as user friendly as possible.
- Another file format, like GraphViz.
- Directed graphs.

10.6 | Handing in

For both part one and part two, make a pull request from compare: development into base: master and confirm that CircleCI accepts your pull request. For the second deadline, also write a report containing the decisions you made while you were making this program.

IV | Appendix

A | Git commands

Command	Purpose
<code>git --help</code>	Provides you with the documentation of git.
<code>git add <filename></code>	Adds the provided files to the stage, ready to be committed. You can even list multiple files at once, to add all of those files to the stage.
<code>git add -A</code>	Adds all the tracked and untracked files to the stage.
<code>git reset <filename></code>	Removes the specified file from the stage; Undoes an add.
<code>git checkout <filename></code>	Resets the contents of the specified file to the last commit. Undoes progress made since the last commit.
<code>git checkout -b <branch name></code>	Creates a <i>new</i> branch with the specified name. Also switches the local repository to the new branch.
<code>git checkout <branch name></code>	Switches the local repository to the specified branch.
<code>git checkout <commit></code>	Resets all progress in the working directory to the state of the specified commit. This should be used with caution, as this does not remove the commits made after the specified commit; your workspace only seems to have those commits reverted.
<code>git commit</code>	Commits all staged files and opens a editor for the commit message.
<code>git commit -m <message></code>	Commits all staged files and uses the specified commit message.
<code>git commit -a</code>	A combination of <code>git add -A</code> and <code>git commit</code> .
<code>git rm <filename></code>	Removes a file from both the working directory and the local repository.
<code>git push</code>	Pushes commits in the local repository to the remote repository currently set as the upstream, to the branch corresponding to the branch you're locally tracking.
<code>git push <remote></code>	Pushes commits in the local repository to the indicated remote repository. Fails if the remote and local repositories do not have matching branches.
<code>git push <remote> <branch></code>	Pushes commits in the local repository to the indicated remote repository, to the specified branch. This branch doesn't necessarily have to correspond with the branch you're locally tracking.
<code>git push --set-upstream <remote> <branch></code>	Pushes commits in the local repository to the indicated remote repository, to the specified branch. Also sets the upstream to the specified remote/branch combination.
<code>git fetch</code>	Fetches remote commits from the upstream.
<code>git merge</code>	Merges fetched commits into the working directory.
<code>git merge <branch></code>	Merges the specified branch into the the branch you're currently tracking. This creates a new commit. Also works for multiple branches.
<code>git pull</code>	A combination of <code>git fetch</code> and <code>git merge</code> .
<code>git pull <remote></code>	Pulls remote commits from the specified remote repository.
<code>git pull <remote> <branch></code>	Pulls remote commits from the specified remote repository, and from the specified branch.
<code>git status</code>	Summarises the status of the local repository.
<code>git log</code>	Produces the commit history and the messages attached.
<code>git log --graph</code>	Produces the commit history in a graph-like fashion, which makes it easier to identify the merge commits.
<code>git log --oneline</code>	Produces the commit history in a more compact manner.
<code>git remote add <name> <url></code>	Adds a new remote repository with the specified url.
<code>git remote rename <oldname> <newname></code>	Renames a remote.
<code>git remote remove <name></code>	Deletes the specified remote.
<code>git remote set-url <name> <url></code>	Changes the url of the specified remote.
<code>git revert <commit></code>	Reverts the changes contained in the specified commit. Also commits these new changes.
<code>git revert -n <commit></code>	Reverts the changes contained in the specified commit. Does not commit these new changes, but only applies the changes to the working directory. The commit identifier can be found through the <i>log</i> command. Alternatively, use <code>HEAD~3</code> for the fourth last commit of the local repository. Similarly <code>master~5..master~2</code> to revert the sixth last commit in <i>master</i> through the third last commit in <i>master</i> (closed interval).
<code>git config [--local] <name> <value></code>	Writes configuration variables to the local repository settings. The <code>--local</code> option is optional, as this is the default and must only be used if the default has been modified. Alternatives are <code>--global</code> , for a global configuration per user on the machine; <code>--system</code> for system wide configuration, which applies to all users of the machine. There exist many different parameters that can be set, and these parameter should be used as the <i>name</i> argument. The value to which the parameter should be set is the <i>value</i> argument. The most noteworthy parameters are: <ul style="list-style-type: none"> <code>user.name</code> sets the name of the user, to be attached to commits. <code>user.email</code> sets the email of the user, to be attached to commits. Additionally, GitHub uses this address to identify the account, even though you can authenticate with a different account.