Programming Assignment 0
Java, Unix, Gradle, and Git

To run the assignments in the course, you need basic familiarity with Java, Unix, Gradle, and Git. This assignment gives a basic introduction to these tools.

Depending on your experience, you might already be familiar with some or many of the topics of this assignment. You can skip any section that practices things you are already familiar with. If you have questions, don't hesitate to ask at the course forum.

You need a Unix command line to do this assignment. You can either use your own machine (Mac, Linux, or Linux subsystem for Windows), or one of the student lab machines with Linux.

There are student lab machines with Unix in the following rooms:
`venus, mars, jupiter` (north part of E-house basement)
`hacke, panter, lo, val, falk, varg` (south part of E-house basement)
`alfa, beta, gamma` (2nd floor, E-house)
`backus` (entry level, Math house)

1 The Unix filesystem

Can you perform the following tasks using the Unix command line, without looking in a manual?

1. Open a terminal window and go to your home directory.
2. Print the full path name of your home directory.
3. Find out what files there are in the current directory.
4. Get help for a command by reading its manual. For example, can you find out what the `-t` option does for the `ls` command?
5. Go to the parent directory. What files are there and which access rights do they have?
6. Go to the `/bin` directory. Do you recognize any of the files there?
7. Go back to your home directory and create two new subdirectories `adir` and `bdir`.
8. Create a new file `afile` in the `adir` directory. (You can use the `touch` command to create new empty files.)
9. Copy the file `afile` from `adir` to `bdir`.
10. Copy the directory `adir`, including its contents, to a new directory `cdir`.
11. Rename the file `afile` in `bdir` to `bfile`.
12. Rename the directory `cdir` to `ddir`.
13. Move the file `bfile` in `bdir` to `ddir`.
14. Move the directories `adir` and `bdir` into the directory `ddir`.
15. Remove the directory `ddir` including all its contents.
16. If you did everything right, all the files and directories you created in this exercise should now be gone again.

If you are sure you can do all these tasks without looking in a manual, you can skip the rest of this section and go to Section 2. If not, read the rest of this section, then go back and do the tasks above.

1.1 Opening a terminal window

If you don't know how to open a terminal window on your operating system, search on your favorite search engine with a query like this:

```
open terminal window <operating system>
```

1.2 Navigation

Play around with the following commands to understand how you can navigate in the Unix file system: *Note!* If you don't know how to type the tilde character (`~`) on your platform, google this.

`pwd`	Print working directory.
`cd`	Change directory to your home directory.
`cd dir`	Go to the *local* directory `dir`, relative to the current directory.
`cd /usr/bin`	Go to the directory with the *absolute* path `/usr/bin` (not relative to current directory).
`cd ..`	Go to the parent directory.
`cd ~/foo`	Go to the directory `foo` in your home directory.
`ls`	List all files in the working directory. Files starting with a period, like `.gitignore`, are not listed.
`ls -a`	List all files in the working directory, including those starting with a period.
`ls -l -h`	List files with detailed information like size, change time and access rights.
`man ls`	Show the manual for the `ls` command. Type space to go to the next page. Type `b` to go to the previous page. Type `q` to quit.

1.3 Manipulation

Play around with the following commands to understand how you can manipulate files and directories. *Note!* There is no undo in Unix! **Be careful** not to delete or overwrite important files.

`mkdir d`	Make (create) directory `d`.
`touch f`	Create a new empty file named `f`. The `touch` command updates the last modified timestamp of a file, but if the file did not exist before, it is created.
`cp dir/f1 f2`	Copy file `dir/f1` and name the copy `f2`. The copy will be created in the working directory, the original is in subdirectory `dir`.
`cp f ..`	Copy file `f` to its parent directory.
`cp ../f .`	Copy file `f` in the parent directory to the working directory.
`cp -r d1 d2`	Recursively copy directory `d1` into directory `d2`; copies `d1` and all its content.
`rm f`	Remove file `f`.
`rmdir d`	Remove directory `d`; however, works only if `d` is empty.
`rm -r d`	Recursively remove directory `d`; deletes `d` and all its content.
`mv x y`	If `x` is a file and `y` a directory: moves `x` into `y`.
`mv x y`	If `x` is a file and `y` does not exist: renames `x` to `y`.
`mv x y`	If `x` and `y` are files: renames `x` to `y` – the old `y` is overwritten.
`mv x y`	If `x` is a directory: rename it to `y` if `y` does not exist yet, otherwise move it into `y` (assuming `y` is a directory), otherwise the operation is aborted with an error (because `y` is an already existing file and not a directory).

1.4 Further reading

If you need further help in using Unix commands, you can read, for example:

`https://linuxcommand.org` Look at the "Learning the Shell" chapter, and in particular the subsections on Navigation, Looking Around, and Manipulating Files.

Introduktion till LTH:s Linuxdatorer, Per Foreby, 2020. In Swedish.
`http://www.ddg.lth.se/perf/unix/unix-x.pdf`

1.5 Check your proficency

Now check your proficiency with Unix commands by going back to the beginning of section 1 and do the tasks there.

2 Editing text files

You should be able to use a text editor like VS Code, Emacs, Vim, Gedit, or Nano to edit code and other descriptions as text files. All of these editors run on all major platforms (Windows, Linux, Mac), and most are preinstalled at the LTH student computers. If you have no idea which to run, we suggest that you try VS Code. It is very well known and easy to use. It also has syntax highlighting for the JastAdd tool we will use later in the course.

2.1 Create a small Java program by using a text editor

Use the text editor to construct a java program `Hello.java`:

```
public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello!");
  }
}
```

Make sure the file contains the correct code by typing

```
less Hello.java
```

in the terminal window. The `less` command lists the contents of a file, and you can move forward and backward one page at a time using space and `b`, just like in the `man` command. You can also use the arrow keys to move forward and backward a line at a time. Type `q` to quit from the `less` command.[1]

3 Java from the command line

You need to be able to compile and run Java programs from the command line.

Can you do the following tasks without looking in a manual?

- Implement two Java classes, `A` and `B` in a package `p`, where `B` contains a main method and imports `A`. The main method prints `Hello!` on the standard output and the `toString()` value of an `A` object.
- Compile all classes.
- Run the main method of `B`.

[1] If you wonder why it is called `less`: it is a more powerful and faster variant of the original unix `more` command.

- Pack the classes into an executable Java Archive called `hello.jar` (you have to create a manifest).
- Execute `hello.jar`.
- Add a `while (true) { System.out.println("Foo"); }` loop to the main method of `B`; recompile it. If you execute it, how can you abort?
- Add a string argument to `B` so that it prints `Hello` *arg*`!` instead of only `Hello!`, where *arg* is the argument you give on the command line when you run `B`.

Hints: Classes must be in directories named like their package; their first code line should declare the package name. To compile and run them, your working directory has to be the parent directory of the package hierarchy.

If you are sure you can do all these tasks without looking in a manual, you can skip the rest of this section. Otherwise, read through the rest of this section and then go back and do the tasks above.

3.1 Compiling and running Java programs

Here are some commands for compiling and running Java programs from the command line.

`javac C1.java C2.java`	Compile source files `C1.java` and `C2.java` to class files `C1.class` and `C2.class`. The class files contain bytecode for the Java Virtual Machine.
`javac p/*.java`	Compile all java source files in the package `p`.
`javac -d dir p/C.java`	The generated class files are placed in the directory `dir`.
`javac -cp d1:d2 C.java`	The compilation classpath (`-cp`) is extended with the directories `d1` and `d2` so that the compiled class `C` can use classes in `d1` and `d2`. By default, the classpath includes the working directory and the standard Java library with `lang`, `util`, and so on.
`java p.C`	Execute the main method of class `C` in package `p`.
`java p.C just three args`	Execute `p.C` with string arguments `"just"`, `"three"`, and `"args"`.
`java -cp d1:d2 p.C`	The execution classpath (`-cp`) is extended with directories `d1` and `d2` so that the executed class `p.C` can use classes in `d1` and `d2`.
`java -jar prog.jar`	Execute the main method specified in the manifest of the Java Archive `prog.jar`.
`ctrl-c`	(while program runs) Abort execution of running program.

Note that the compilation command (`javac`) takes files as arguments, like `p/C.java`, whereas the execution command (`java`) takes classes as arguments, like `p.C`.

3.2 Jar files and classpaths

Java programs are packaged and distributed as JAR (**J**ava **Ar**chive) files. We will now create a JAR file that will contain the `A` and `B` classes defined before. A JAR file requires a Manifest file that specifies, for example, which class is the main class of the JAR file. We will start by creating the Manifest file.

Create a file `MANIFEST.MF` and add the following code to it, ending with a blank line:

```
Manifest-Version: 1.0
Main-Class: p.B
```

We can see that `B` in the package `p` is specified as the main class. The command for creating JAR files is `jar`. The pattern for using it is as follows.

```
$ jar cfm jar-file manifest-file input-file(s)
```

For example, we can create a JAR file `HelloJar.jar` as follows, using the Manifest file that we defined earlier.

```
$ jar cfm HelloJar.jar MANIFEST.MF p/A.class p/B.class
```

Note that we only include the class files, and not the source code (`A.java` and `B.java`). A JAR file can contain source code as well, but that is not necessary. Having the JAR file, it can be executed as follows.

```
$ java -jar HelloJar.jar
```

The command `java` starts the Java virtual machine (JVM) which runs a program by loading class files and calling the `main` method of the main class.

The *classpath* specifies where the JVM should look for the class files. Specifying the classpath may be required when several different libraries are used that are located in different JAR files. The classpath can be given using the flag `-cp` as the following illustrates.

```
$ java -cp lib1.jar:lib2.jar:. MainClass
```

The classpath is given as a list of colon-separated JAR files and directories. In this case, the JVM will look for class files in `lib1.jar`, in `lib2.jar`, and in the current working directory; the dot (`.`) means the current working directory. The standard classpath is the current working directory, which is equivalent to the following.

```
$ java -cp . MainClass
```

Often, it is required to include the library files when compiling the source code as well. For example, suppose we have a class `MainClass` that uses classes from the JAR files `lib1.jar` and `lib2.jar`. Then when compiling the `MainClass.java` file, we need to include the JAR files in the classpath as follows.

```
$ javac -cp lib1.jar:lib2.jar MainClass.java
```

3.3 Check your proficiency

Now check your proficiency with creating, compiling and running Java programs by going back to the beginning of section 3 and do the tasks there.

3.4 A note on Java versions

New versions of the Java language are issued rather frequently. The labs have been tested on Java 11, Java 17, and Java 21. You can find out what version your computer is running by the command:

```
$ java --version
```

If you get something like

```
openjdk 21.0.4 2024-07-16
```

this means you are running Java 21. If you are running some other version than Java 11, 17, or 21, there could be problems, and you might need to switch Java version. On Mac and Linux, you can use the tool `https://sdkman.io` to easily switch between different Java versions.

Even if you are running on Java 21, you will not be able to use language constructs that are newer than those in Java 8 in the labs. This means, for example, that you cannot use the `var` construct that was introduced in Java 11. The reason is that we are using some tools that so far only support Java 8 code.

4 The build tool Gradle

As a project grows larger, there may be many commands that need to be run in order to compile, test, create jar files, etc. Running these commands manually after each change of the source code would quickly become both tedious and error prone. Instead, software projects use *build tools* that document and automate the commands to run. Examples of common build tools are Make, Ant, and Gradle. In this course, we will use *Gradle*.

The Gradle build tool is implemented in the language Groovy, which compiles to Java bytecode. We will run Gradle by using the script `gradlew` (or `gradlew.bat` on Windows). When this script is run the first time, the Gradle tool (a jar file) is downloaded from the web. This way, you don't need to manually install Gradle on your computer.

Gradle is a very powerful build tool. Here are some things it can do:

- automate common tasks like compiling, producing jar files, and running tests
- support different languages by the use of *plugins*. There is, for example, a Java plugin.
- incremental build, i.e., avoid building things that are already up to date. For example, if you run a build, but did not change any source files since your last build, then no compilation or testing will be run.
- automatic download of *dependencies*, i.e., appropriate versions of library files that your project depends on. These files are downloaded from global repositories like Maven and Ivy.

4.1 Gradle basics

To understand the basics of how Gradle works, download `A0-JavaGradleProject.zip` from `https://cs.lth.se/edan65`, and unzip it. This is an empty Java project, set up to build with Gradle. In the project you find the following files and directories:

`build.gradle`	The Gradle build script.
`gradle/`	Files for running Gradle.
`gradlew`	shell script for running Gradle on Unix.
`gradlew.bat`	batch script for running Gradle on Windows.
`src/main/java`	where your main Java code should be placed.
`src/test/java`	where your JUnit test code should be placed.

When you run Gradle, it reads in the file `build.gradle`. This file is called the *build script* and it contains rules for how to build the project. Take a look at `build.gradle`. The only thing it contains is the following lines of code that says that the Java plugin should be used:

```
plugins {
    id "java"
}
```

The Java plugin defines rules for building Java projects. It defines a number of *tasks*, i.e., pieces of work. You can find out which tasks are supported by running the command[2]

```
./gradlew tasks --all
```

[2]The first time you run the `./gradlew` command, it will download the Gradle jar file from the web, which may take a short while. The Gradle jar file will be placed in a directory `.gradle` in your home directory.

As you see, there are quite a few tasks. We will only discuss the following:

`build`	compiles any source and test files, creates a jar file, and runs any test files
`compileJava`	only compiles the main Java code (not the test code)
`compileTestJava`	only compiles the test code (the JUnit tests)
`clean`	removes the generated files (class files, jar files, etc.)

Try out these tasks. For example, run

```
./gradlew build
```

Note that all generated files are placed in a generated directory `build`. Since there are no source files yet in this project, there will in this case be no generated class files, but a jar file will be generated, although it contains nothing but a manifest. The jar file is named `A0-JavaGradleProject.jar` after the project directory.

Now run the `clean` task, and note that the `build` directory with all the generated files is removed.

```
./gradlew clean
```

Tasks depend on each other in a directed acyclic graph. When a task is called, and everything it depends on is unchanged, it does not have to be run again. Instead, it is reported to be `UP-TO-DATE`. This is called *incremental* building: after changes, Gradle rebuilds only the parts of the project that are affected by the change.

As a simple example of incremental building, run the `clean` task (in case you did something else after your last `clean`), and then run the `build` task twice. The second time you run `build`, everything is already up to date, and no tasks are run.

```
./gradlew clean
./gradlew build
./gradlew build
```

Gradle keeps its internal information in a directory `.gradle` in your home directory, and in a directory `.gradle` in each project. Any dependencies (library jar files, etc.) downloaded by a build will be stored in the `.gradle` directory in your home directory, so that they can be shared by all your Gradle projects. The Gradle tool itself is also stored there.

4.2 Experiment with Gradle on some Java code

Add some Java code to the project. For example, you could add a package `p` with a class `A` whose main method prints `Hello`. The code should be placed in the `src/main/java` directory. You will need to create the `src/main/java/p` directory and place the `A.java` file inside it. Run a build (`./gradlew build`). Note where Gradle places the classes and the jar file (`build/classes/java/main` and `build/libs`, respectively). You should now be able to run your code by either

```
java -cp build/classes/java/main p.A
```

or

```
java -cp build/libs/A0-JavaGradleProject.jar p.A
```

4.3 Experiment with adding a test case

We will now show how you can run JUnit tests from Gradle. JUnit is a framework for automated testing of Java code. In the labs, we are using JUnit version 4.13.2.[3]

First you need something to test. Edit your class `A`, and add a static method `m` that simply returns a constant, say 5. Now create a test class `TestA` with a JUnit test method that tests that `m` works correctly:

```
package p;

import org.junit.*;
import static org.junit.Assert.*;

public class TestA {
        @Test
        public void testm() {
                assertEquals(5, A.m());
        }
}
```

Gradle will only recognize tests if they are placed in the `src/test/java` directory, so the above test class should be placed at the location `src/test/java/p/TestA.java`.

The imports used in the test code refer to entities in the JUnit library. To compile the tests, Gradle needs access to this library. To handle this, you declare a *dependency* in the `build.gradle` file. More precisely, when compiling tests, there is a dependency on a particular version of the JUnit library. You also need to state in what global repository Gradle can find the dependencies (i.e., the libraries). You do this by adding the following code to the `build.gradle` file:

```
repositories.mavenCentral()

dependencies {
    testImplementation "junit:junit:4.13.2"
}
```

Here, `testImplementation` is a so called *dependency configuration* defined by the Java Gradle plugin, and which is used by the task that compiles the tests (`compileTestJava`).

Now, when you run `./gradlew build`, Gradle will locate the Maven central repository, and find the JUnit library jar file (version 4.13.2), and download it to your computer (placing it in the `.gradle` directory in your home directory). It will then compile the test code, using this jar on the class path, and it will also run the tests.

Check that you can get this to work, and that the test case succeeds (the build should be reported as successful).

Now change the code so that the test will fail. For example, let the method in `A` return 4 instead of 5. What happens when you build again? Where can you find a report of the test results?

Hint: you can add the `-i` (short for `--info`) to the Gradle command to get more information about test failures.

[3] If you are not familiar with JUnit, or this particular version, read through the following introduction to JUnit: `https://fileadmin.cs.lth.se/cs/Education/EDAN65/misc/junitForEDAF45.pdf`. This introduction was originally written for another course and includes some details on how to use JUnit from Eclipse. Other IDEs have similar support.

5 Use Git for version control of your code

Git is a version control system that allows you to:

- work iteratively, *committing* (saving) each stable change of the code as a new version. This is very useful, for instance, if you mess up and need to restore the previously working version.
- easily collaborate with others. You can *push* your commits to a common repository, from which your partner can *pull* them, and vice versa. You can easily see what your partner has changed, and you can merge your changes if you are both changing the same parts of the code.

For the labs, you will use a Git repository provided by a server at the department. Both you and your partner should commit to the repository several times for each lab. See the course web for details.

To work with the labs, you *clone* the server repository once to your computer, and you can then start working, saving your work, and collaborating with your partner by doing *commit*, *push*, and *pull*. You should be familiar with the following basic Git commands and concepts:

`git clone` *server-loc*	copies the repository located at *server-loc* to your current directory
`git commit -am "`*msg*`"`	commits your changes to your local copy of the repo
`git push`	pushes up your new commits to the server
`git pull`	pulls down other new commits to your local copy of the repo
`git add` *newfiles*	puts the files *newfiles* under version control
`git status`	show the current status of your local copy of the repo
`.gitignore`	a file that specifies what files git should ignore

Other useful commands are:

`git rm` *files*	version-aware removal of *files*
`git mv` *file1 file2*	version-aware rename of *file1* to *file2*
`git diff` *files*	shows the diffs between your edited *files* and the latest commit
`git stash`	moves all your changes (since the latest commit) to a hidden *stash*
`git stash apply`	applies the stashed changes again

If you are familiar with these Git commands, you can skip the rest of this section. The official documentation for Git is here: `https://git-scm.com`

5.1 Getting started with the Git repository

First, check that you have Git installed.

```
git version
```

If not, consult `https://git-scm.com` for how to install git.

If you have not used Git before on your computer, you should also do a First-Time Git Setup, in order to set what username and email Git should attach to your commits, and what default editor Git should start if it wants to prompt you with a message. See `https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup`.

In the labs, we will use repos at the department git server `https://coursegit.cs.lth.se`. There are two ways to clone repositories: using SSH or using HTTPS, and some preparations are needed for each of them:

HTTPS You will need your coursegit username and a coursegit PAT (Personal Access Token) in order to clone using https. To find out your username, first login to `https://coursegit.cs.lth.se` using your LUCAT/STIL credentials. Then, in the avatar menu in the top right corner, you can see your username, written as @username. To get a PAT, follow the instructions at `https://coursegit.cs.lth.se/-/profile/personal_access_tokens`. Copy the PAT code to a safe place, and use it as the password when cloning.

SSH To use ssh, you need to add an SSH key to coursegit. Follow the instructions at `https://coursegit.cs.lth.se/-/profile/keys`.

Next, create a directory on your computer where you want to place your work, and clone your repository (replacing `yyyy` and `reponame` with the year and name of your repository). For example, if you are using HTTPS, write:

```
mkdir edan65
cd edan65
git clone https://coursegit.cs.lth.se/edan65/yyyy/reponame
```

A subdirectory `reponame` will be created. Initially, it will contain a file `README.md` and one subdirectory for each lab (`A0, A1, ...`), each with a local `README.md` file. The `README.md` files are all in the markdown format. Check that your directory contains these files:

```
cd reponame
ls
```

5.2 Change files, commit and push

To get started with changing files, edit the `README.md` file to add your names to it, then commit it, and push it up to the server. Do `git status` between the commands, so you see what happens:

```
edit the README.md file
git status
git commit -am "Improved README"
git status
git push
git status
```

Note! the option `-a` tells git that it should commit all modified files that are under version control. The option `-m` with the argument `"Improved README"` provides the commit log message.

Note also that the commit gets a unique identifier, e.g., something like `7dffdae5`.

Browse to your repository on the server, and verify that the contents were pushed: Select `Code->Repository` in the left-column menu to see the files in the repository, and `Code->Commits` to see a list of commits.

5.3 Add a new file, and put it under version control

If you create new files that you would like to include in the repository, you need to explicitly add them to git. You do this by creating the file, adding it to git, then committing, then pushing. Again, do `git status` between each command, so you understand what goes on. When experimenting, we suggest you do your changes in the subdirectory for this assignment: `A0`.

```
go the the A0 directory to add an example file
edit a new file, say example.txt
git add example.txt
git commit -am "Added example file"
git push
```

Note that it is easy to forget to do `git add` on new files that you want in the repository. If you do `git status` you will see what files you have not added so far.

5.4 The .gitignore file

Often, the tools you use will create files in your directory that you *don't* want to version control. Examples include backup copies of textfiles, generated files like class files and jar files, etc.

As a general rule: *don't put generated files under version control.* They will give you messy merge conflicts, and will make your repository grow in size very quickly, taking up unnecessary resources and slowing down your work.

In your repo, there is a file `.gitignore` with rules for what files Git should ignore.

Here is an example of a very small `.gitignore` file:

```
# java (ignore .class files generated from Java)
*.class
#
# Mac OS (ignore settings files generated by Mac operating system)
.DS_Store
#
# Ignore the file hidden.txt
hidden.txt
```

Take a look at the `.gitignore` file in your repository. It has been prepared to work for the course, ignoring suitable files. But feel free to update it with more rules during the course.

```
more .gitignore
```

5.5 Collaboration and merging

When you collaborate with your partner on the same repository, you will need to pull down your partner's changes now and then. If you have changed the same files, you will need to merge them. These things are simple to do with git. Try out a simple scenario at first:

1. You change a file, commit, and push the changes to your common repository on the server.
2. Ask your partner to pull the change to his/her computer.
3. Ask your partner to change a file, commit, and push.
4. Now you pull the new changes.

Now let's look at a scenario where you both edit at the same time, and you might get merge conflicts.

1. Ask your partner to change a file, commit, and push.
2. Now, before pulling, you also make a change to the same file, at the same line. Then commit, and push.
3. Note that you cannot push, because your commits are not changes based on the latest version on the server.
4. You need to first do `git pull`. Your partner's changes will then be merged with what you have in your repository. An automatic merge commit will be created.
5. Note that if you edited the same file, git will try to merge the two versions of the file. If you edited on the same line there will be merge conflicts that you will have to resolve manually. Git will insert some markers in the file, so you see what changes came from what version.
6. Fix any merge conflicts by editing the file. Then you can commit again, and push.

5.6 Push often and clean

When collaborating, it is important that each commit and push represents a clean state of the project. For example, all the code should compile without errors, and all tests should run.

In order to collaborate in a good way, it is good to commit, push, and pull often:

- Before starting on some new work, check that your repository doesn't contain uncommitted work (use `git status`), and do a `git pull` so that you start working on the latest version. Check that the new version you pulled down actually works (compiles and tests) before starting. This should have been checked by the person doing the commit – but they might have missed something.
- When doing work, split it down into small clean pieces. For example, make one little thing work, or work better than before. Check that it compiles and tests before committing and pushing. Commit each little piece of clean new/improved functionality.
- Commit often. That way, if you mess up, it is easy to back up to the latest stable commit, e.g., using `git stash`.
- Push often. That way your partner has to merge instead of you. And the merges will be smaller and easier for your partner than if you wait.
- Pull often. That way the changes that your partner made will be smaller and easier to merge than if you wait.
- Communicate with your partner. That way you can avoid unpleasant surprises.

6 Further reading

To get quick help, use a search engine like Google. Often you will find useful answers at Stack Overflow (`https://stackoverflow.com`).

Additional resources:

- Java cheat sheet from Princeton:
  `https://introcs.cs.princeton.edu/java/11cheatsheet/`
- Oracle's Java tutorial: `https://docs.oracle.com/javase/tutorial/java/index.html`
- Java standard base library documentation: `https://docs.oracle.com/en/java/javase/11/docs/api/java.base/module-summary.html`
- (Alternatively, look at the older library for Java version 8, which might be more convenient to navigate: `https://docs.oracle.com/javase/8/docs/api/index.html`)
- About running `java` and `javac` from the command line:
  ```
  man java
  man javac
  ```
- JAR tutorial: `https://docs.oracle.com/javase/tutorial/deployment/jar/index.html`
- Gradle documentation: `https://docs.gradle.org/8.5/userguide/userguide.html`
- User guide for the Java plugin for Gradle: `https://docs.gradle.org/8.5/userguide/building_java_projects.html`
- JUnit 4 documentation: `https://junit.org/junit4/`
- Git documentation: `https://git-scm.com`