Second Brain

Object-Oriented Programming
2nd Project, version 1.0 – 2025-05-12

Important remarks

Deadline until 23h59 (Lisbon time) of June 1st, 2025. Submissions are only possible via Mooshak. Late submissions will suffer a penalty of 1 point (out of 20) for each set of 8 hours of delay (for example, a submission delivered 10 hours after the deadline will suffer a 2-point penalty).

Team This project is to be MADE BY GROUPS OF 2 STUDENTS.

Deliverables: Submission and acceptance of the source code to **Mooshak** (score > 0). See the course website for further details on how to submit projects to Mooshak.

Recommendations: We value the documentation of your source code, as well as the usage of the best programming style possible and, of course, the correct functioning of the project. Please carefully comment both interfaces and classes. The documentation in classes can, of course, refer to the one in interfaces, where appropriate. Please comment on the methods explaining what they mean and defining preconditions for their usage. Students may and should discuss any doubts with the teaching team, and with other students, but may not share their code with other colleagues. This project is to be conducted with full respect for the Code of Ethics of NOVA University, Lisbon, available on the course website.

1 Development of the application *Second Brain*

1.1 Problem description

The goal of this project is to develop an application that supports the creation of a second brain - a personal knowledge management system that can be used by students, researchers, and other knowledge workers. You can think of this as a sort of personal Wikipedia. As you learn new things, you create notes about what you have just learned. Crucially, you create links among these notes, as you establish connections between the knowledge pieces. The rationale is that, in the future, you can revisit your notes and navigate through them in meaningful ways. The system to implement is inspired and adapted from Niklas Luhmann's Zettelkasten, a widely popular personal knowledge management system.

Zettelkasten means *card file*, as in *a box with paper cards*. The key element of this system is the notion of note. Luhmann created a system for writing, indexing, and storing notes in such a way that he could easily find notes and discover connections among them even years after they were originally written. Whenever he read something interesting, learned something potentially useful, or had an idea, he created a note for it. Using his note-taking system, he became one of the most prolific social scientists in the 20th century by leveraging the tens of thousands of notes he wrote to create ground-breaking scientific contributions.

Your goal is to create a prototype of a digital Zettelkasten. A **note** has a **title**, which is used as a unique identifier (a String which may contain spaces), a **date** of its creation, the **date of its last edition**, and, with the exception of **reference notes**, a **content**. It also includes **links** to related notes, much like those you find in a web page, such as in a Wikipedia entry. In this system, we will consider two kinds of links - **internal** and **external** links. Internal links are links to other notes. External links are links to external resources that are referred to in notes, but are not notes themselves (e.g., a URL).

There are three kinds of notes to consider:

- **literature notes** are notes similar to those one would write about a particular book quote, maybe directly in the book's margins next to the highlighted quote. These notes include the **highlighted quote** and any **comment** the user would like to keep about the quote. This **comment** is a text that may include **links to other notes**. Additionally, literature notes have a list of **reference notes** that reference them.

- **permanent notes** capture stand-alone ideas and summarise information you just learned, or a thought that occurred to you. For all practical matters, this is captured just like the **comment** we just discussed for literature notes. It is common to analyse the summaries in permanent notes and figure out how they fit into your existing knowledge, leading to new **links to other notes** (just like we do for the **comment** in literature notes). Additionally, permanent notes have a list of **reference notes** that reference them. Unlike literature notes, permanent notes keep a history of update dates, rather than just the last update date.

- **reference notes** help you group related information so that you can find it at a later stage. A common example of referencing is the usage of the hashtag (♯) on social networking platforms. You can do the same with the notes system. Whenever you tag a note, that note becomes referenced in a reference note that represents all notes with that tag.

Table 1 summarises the information held by the different note kinds.

A typical system will have thousands of notes, but each note has a small number of links to other notes. Reference notes usually have a small number of referenced notes. These systems are typically used for several years, so you can expect to have notes introduced on hundreds or thousands of dates.

Table 1: Information held by each kind of note

	Permanent Note	Literature Note	Reference Note
Note id	x	x	x
Note creation date	x	x	
Note last update date	x	x	
Note update history	x		
Note's comment	x	x	
Links to other permanent and literature notes	x	x	x
Links to reference notes	x	x	
Artifact title		x	
Artifact author		x	
Artifact date		x	
External link to artifact		x	
Quote from the artifact		x	

2 Commands

In this section, we present all the commands that the system must be able to interpret and execute. In the following examples, we differentiate `text written by the user` from the feedback written by the program in the console. You may assume the user will make no mistakes when using the program other than those described in this document. In other words, you only need to take care of the error situations described here, in the same order as they are defined.

Commands are case-insensitive. For example, the `exit` command may be written using any combination of upper and lowercase characters, such as `EXIT`, `exit`, `Exit`, `exIT`, and so on. In the examples provided in this document, the symbol ↵ denotes a line change.

The program must write the message Unknown command. Type help to see available commands. to the console if the user introduces an unknown command. For example, the non existing command `someRandomCommand` would have the following effect:

```
someRandomCommand↵
Unknown command. Type help to see available commands.↵
```

If there are additional tokens in the line (e.g., a parameter for the command you were trying to write), the program will also try to consume them as commands. So, in the following example, `someRandom Command` would be interpreted as two unknown commands: `someRandom` and `Command`, leading to two error messages.

```
someRandom Command↵
Unknown command. Type help to see available commands.↵
Unknown command. Type help to see available commands.↵
```

Several commands have arguments. Unless explicitly stated in this document, you may assume that the user will only write arguments of the correct type. However, some of those arguments may have an incorrect value. Therefore, we must test each argument precisely in the order specified in this document. In their description, arguments will be denoted **`with this style`**, for easier identification. Also, assume by default that any String arguments in the commands are case-sensitive. So, "Java Iterator" and "java iterator" would be two different note titles.

2.1 `exit` command

Terminates the execution of the program. This command does not require any arguments. The following scenario illustrates its usage.

```
exit↵
Bye!↵
```

This command always succeeds.

2.2 `help` command

Shows the available commands. This command does not require any arguments. The following scenario illustrates its usage.

```
help↵
create - creates a new note↵
read - reads a note↵
update - updates a note↵
links - lists all links in a note↵
tag - tags a note↵
untag - untags a note↵
tags - lists all tags in alphabetical order↵
tagged - lists all notes with a specific tag↵
trending - lists the most popular tags↵
notes - lists all notes of a given type last edited within a given time interval↵
delete - deletes a note↵
help - shows the available commands↵
exit - terminates the execution of the program↵
```

This command always succeeds. When executed, it shows the available commands.

2.3 `create` command

Creates a new note in the system. The command receives, as arguments, the **`kind`** of note, its **`date of creation`**, the note's **`unique id`**, and the note's **`content`**.

Notes can be created as **`permanent`** or as **`literature`** notes. You may assume that the user will always create notes of a valid kind (either **`permanent`** or **`literature`**). Reference notes are **NOT** created with this command (see the **`tag`** command for further details). The **`content`** of a note may encode links to other notes. These links are encoded as Strings starting with **`[[`** and ending with **`]]`**. Permanent and literature notes also store their creation date and last update date. For the sake of this project consider time does not go back. So, once you make a note on **`2025/05/05`**, it is impossible to create or update notes on previous dates (e.g., on **`2024/12/25`**). All subsequent notes must be on the same or a posterior date.

Let us start with creating **`permanent`** notes. When successful, the note (and possibly any new linked notes) is created and the program outputs the feedback message (Note <id> created successfully with links to <n> notes.). The permanent note history is also initialized with the creation date. In the example, we first create a new note about **`Robert Baratheon`** being the seventeenth ruler of the Seven Kingdoms and the first king of the Baratheon dynasty. Then, we create a second note on how **`Robert Baratheon married Cersei Lannister`**, to strengthen an alliance with House Lannister. Here, we create three links: one to **Robert Baratheon** (the note we just created), one to **Cersei Lannister**, and another to **House Lannister**. This has the side effect of creating a permanent note for **Cersei Lannister** and another to **House Lannister**, if those notes do not exist. **When automatically creating these notes, they are created as permanent notes, with the same creation date and with the note id followed by a period as content. For example, the permanent note on Cersei Lannister would have the content "Cersei Lannister." and creation date 2025 05 06.** In any case, the new note creates links to those three notes. These links are unidirectional, in the sense that if a note creates a link from Robert Baratheon to Cersei Lannister, the note on Robert Baratheon **refers to** the note of Cersei Lannister, and the note of Cersei Lannister **is referred to** by the note on Robert Baratheon. So, if you were to list the notes referenced in the comment about Robert Baratheon you would find Cersei Lannister's note, but if you were to list the notes referenced by the comment on Cersei Lannister's note, you would not necessarily find the note on Robert Stark (you only find it if the comment on Cersei Lannister's note happens to refer to the note on Robert Baratheon, which may, or may not, happen). There is no upper limit to the number of notes (of any type) one can create in the system. A third note on **Robert Baratheon brothers**, Renly and Stannis Baratheon, illustrates that the system counts the number of unique links to other notes, when a note contains more than one link to the same note (in this case, the two references to **Stannis Baratheon** which link to the same note).

```
create permanent 2025 05 05↵
Robert Baratheon↵
King Robert Baratheon was the seventeenth ruler of the Seven Kingdoms
and the first king of the Baratheon dynasty.↵
Note Robert Baratheon created successfully with links to 0 notes.↵
create permanent 2025 05 05↵
Robert Baratheon marriage↵
[[Robert Baratheon]] married [[Cersei Lannister]] to strengthen an
alliance with [[House Lannister]].↵
Note Robert Baratheon marriage created successfully with links to 3 notes.↵
create permanent 2025 05 06↵
Robert Baratheon brothers↵
[[Robert Baratheon]] had two brothers: [[Renly Baratheon]] and
[[Stannis Baratheon]]. [[Stannis Baratheon]] declared himself
Robert's rightful heir and assassinated Renly using [[bloodmagic]].↵
Note Robert Baratheon brothers created successfully with links to 4 notes.↵
```

The following errors may occur:

1. If the **date** is invalid, the error message is (Invalid date!).
2. If the **date** precedes the most recent creation or update date in the system, the error message is (No time travelling!).
3. If the system already has a note with a given **note id**, the error message is (<note id> already exists!).

The following example illustrates these error messages.

```
create permanent 2026 02 30↵
Ned Stark survival↵
Ned Stark is Sean Bean's first character ever not to die.↵
Invalid date!↵
create permanent 2023 02 28↵
Ned Stark survival↵
Ned Stark is Sean Bean's first character ever not to die.↵
No time travelling!↵
create permanent 2025 05 06↵
Robert Baratheon↵
King Robert Baratheon was the seventeenth ruler of the Seven Kingdoms
and the first king of the Baratheon dynasty.↵
Robert Baratheon already exists!↵
```

Literature notes require some additional information compared to permanent notes. Just like permanent notes, they include `creation date` (and `last update date`, which is implicitly set to be the same as the creation date, when the note is created, but will be updated with the `update` command), `unique id` and `the note's content`. Both types of notes allow to create references to other notes. Additionally, Literature notes include the `literature work's title` and `author name`, its `publication date`, `URL` and a direct `quote taken from the original text`.

```
create literature 2025 05 06↵
Cersei Lannister quote↵
[[Cersei Lannister]] explains Ned Stark the consequences of being
played by [[Sean Bean]].↵
A Song of Ice and Fire↵
George R. R. Martin↵
1996 08 01↵
https://en.wikipedia.org/wiki/A_Game_of_Thrones↵
When you play the game of thrones, you win or you die.↵
Note Cersei Lannister quote created successfully with links to 2 notes.↵
```

The following errors may occur:

1. If the `creation date` is invalid, the error message is (Invalid date!).
2. If the `document date` is invalid, the error message is (Invalid document date!)
3. If the `creation date` precedes the most recent creation or update date in the system, the error message is (No time travelling!).
4. If the system already has a note with a given `note id`, the error message is (<note id> already exists!).
5. If the `document date` is later than the most recent creation or update date in the system, the error message is (No time travelling to the future!).

The following example illustrates some of these error messages (the others are similar to what we did for permanent notes).

```
create literature 2025 02 30↵
Catelyn Stark on humour↵
Catelyn was a wise woman.  Here is her take on humour.↵
A Song of Ice and Fire↵
George R. R. Martin↵
1996 08 01↵
https://en.wikipedia.org/wiki/A_Game_of_Thrones↵
Laughter is poison to fear.↵
Invalid date!↵
create literature 2026 02 28↵
Catelyn Stark on humour↵
Catelyn was a wise woman.  Here is her take on humour.↵
A Song of Ice and Fire↵
George R. R. Martin↵
1996 08 41↵
https://en.wikipedia.org/wiki/A_Game_of_Thrones↵
Laughter is poison to fear.↵
Invalid document date!↵
create literature 2025 02 28↵
Catelyn Stark on humour↵
Catelyn was a wise woman.  Here is her take on humour.↵
A Song of Ice and Fire↵
George R. R. Martin↵
1996 08 01↵
https://en.wikipedia.org/wiki/A_Game_of_Thrones↵
Laughter is poison to fear.↵
No time travelling!↵
create literature 2025 05 06↵
Cersei Lannister quote↵
[[Cersei Lannister]] explains Ned Stark the consequences of being
played by [[Sean Bean]].↵
A Song of Ice and Fire↵
George R. R. Martin↵
1996 08 01↵
https://en.wikipedia.org/wiki/A_Game_of_Thrones↵
When you play the game of thrones, you win or you die.↵
Cersei Lannister quote already exists!↵
create literature 2025 05 08↵
Winds of Winter publication date↵
The Winds of Winter is the planned sixth novel in the epic fantasy
series [[A Song of Ice and Fire]] by American writer [[George R. R.
Martin]].↵
A Song of Ice and Fire↵
George R. R. Martin↵
2025 12 25↵
https://en.wikipedia.org/wiki/A_Game_of_Thrones↵
Not yet, anyway↵
No time travelling to the future!↵
```

2.4 read command

Prints the contents of a note. The command receives, as argument, the `note id`. When successful, the program will output the feedback message (<note id>: <contents> <number of links> links. <number of tags> tags.). In the example, we print the notes on `Robert Baratheon` and `Cersei Lannister quote`. For the time being, assume there are no tags associated with neither of the notes, so tags count is 0 in both cases.

```
read Robert Baratheon↵
Robert Baratheon: King Robert Baratheon was the seventeenth ruler of the Seven Kingdoms and the
first king of the Baratheon dynasty. 0 links. 0 tags.↵
read Cersei Lannister quote↵
Cersei Lannister quote: Cersei Lannister explains Ned Stark the consequences of being played by Sean
Bean. 2 links. 0 tags.↵
```

The following error may occur:

1. If the `note id` is unknown, the error message is (Note <note id> does not exist!).

The following example illustrates these error messages.

```
read Benioff and Weiss best finale idea↵
Note Benioff and Weiss best finale idea does not exist!↵
```

2.5 update command

Updates a note. The command receives as argument the `note id`, the `update date` and the updated `content`. When successful, the program will update the corresponding note and output the feedback message (Note <note id> updated. It now has <number of links> links.). In the example, we update the permanent note `Robert Baratheon` with information about the love of his life and a few links to other notes. In the output, the <number of links> refers to the number of **different** links held by the note in the end of the update. On the surface, the command works the same way, regardless of the update applying to a permanent or a literature note. However, permanent notes keep a history of the update dates, while literature notes just store the last update date.

```
update Robert Baratheon↵
2025 05 06↵
King Robert Baratheon was the seventeenth ruler of the [[Seven
Kingdoms]] and the first king of the [[Baratheon dynasty]].  The love
of his life was [[Lyanna Stark]].↵
Note Robert Baratheon updated. It now has 3 links.↵
```

The following errors may occur:

1. If the `update date` is invalid, the error message is (Invalid date!).
2. If there is no note with the given `note id`, the error message is (Note <note id> does not exist!).
3. If the `update date` precedes the most recent creation or update date in the system, the error message is (No time travelling!).

The following example illustrates these error messages.

```
update Arya Stark↵
2025 05 06↵
Arya used to say, about her foes, to stick 'em with the pointy end.
She did so a few times, using her [[Needle]].↵
Note Arya Stark does not exist!↵
update Yggrite↵
2013 03 33↵
Yggrite keeps reminding [[Jon Snow]] he knows nothing.↵
Invalid date!↵
update Jon Snow↵
2013 04 28↵
As it turns out, to the surprise of [[Yggrite]], Jon Snow knows a
thing or two.↵
No time travelling!↵
```

2.6 `links` command

Lists all the links from a given note. The command receives as argument the `note id` and prints a list of the linked note ids, one per line, by order of first occurrence in the link within the content of the note. If the note contains no links to other notes, the command prints the message (No linked notes.). In the examples, we start with a note with no links and follow with a note with a few links.

```
links Ramsey Bolton was nice to these people↵
No linked notes.↵
links Arya Stark had a special list↵
Cersei Lannister↵
The Mountain↵
Walder Frey↵
Polliver↵
Joffrey Baratheon↵
Rorge↵
Tywin Lannister↵
Meryn Trant↵
```

The following error may occur:

1. If the note does not exist, the error message is (Note <note id> does not exist!).

The following example illustrates a failed attempt to list the links in `Ramsay Bolton's good deeds`. As it turns out, we have no notes on his good deeds.

```
links Ramsay Bolton's good deeds↵
Note Ramsay Bolton's good deeds does not exist!↵
```

2.7 `tag` command

Adds a new reference to a `reference note` (the tag). The command receives as arguments the `referenced note unique id` and the `reference note unique id` and tags the referenced note with the reference note. If successful, the program outputs the message (<referenced note id> tagged with <reference note id>.). In the example, a few Starks suffer a massive health setback and end up tagged as `♯dead`. In `Catelyn Stark`'s case, her liveliness status turns into complicated on account of being both `♯dead` and `♯undead`. The latter example also illustrates that not all tags have a ♯.

```
tag Ned Stark↵
♯dead↵
Ned Stark tagged with ♯dead.
tag Rob Stark↵
♯dead↵
Rob Stark tagged with ♯dead.
tag Catelyn Stark↵
♯dead↵
Catelyn Stark tagged with ♯dead.
tag Catelyn Stark↵
♯undead↵
Catelyn Stark tagged with ♯undead.
tag Catelyn Stark↵
liveliness = itsComplicated↵
Catelyn Stark tagged with liveliness = itsComplicated.
```

The following errors may occur:

1. If the note's `note id` does not exist in the system, the error message is (Note <note id> does not exist!).
2. If the note identified by `note id` is already tagged with `reference note id`, then the error message is (Note <note id> is already tagged with <reference note id>!)

The following example illustrates these error messages.

```
tag Ramsay Bolton's good deeds↵
generous↵
Note Ramsay Bolton's good deeds does not exist!↵
tag Daenerys Targaryen↵
Mother of Dragons↵
Note Daenerys Targaryen is already tagged with Mother of Dragons!↵
```

2.8 `untag` command

Removes the reference to a referenced note from a reference note. The command receives as arguments the `referenced note unique id` and the `reference note unique id` and untags the referenced note from the reference note. If successful, the program outputs the message (Note <referenced note id> no longer tagged with <reference note id>.). In the example, we remove the reference note `♯alive` from the note `Ned Stark`.

```
untag Ned Stark↵
♯alive↵
Note Ned Stark no longer tagged with ♯alive.↵
```

The following errors may occur:

1. If the note's `note id` does not exist in the system, the error message is (Note on <note id> does not exist!).
2. If the note referenced with `note id` is not referenced with (`reference note id`), the error message is (Note on <note id> is not tagged with <reference note id>.).

The following example illustrates these error messages.

```
untag Ramsay Bolton's good deeds↵
♯myth↵
Note on Ramsay Bolton's good deeds does not exist!↵
untag Littlefinger↵
trustworthy↵
Note on Littlefinger is not tagged with trustworthy!↵
```

2.9 `tags` command

Lists the tags associated to a given note. The command receives as argument the `note id` and lists all the reference notes (tags) associated with it. The tags are listed in alphabetical, case-sensitive order. If there are no reference notes to list, the program should print the message (No tags.).

```
tags Winds of Winter release date↵
No tags.↵
tags Daenerys Targaryen↵
Breaker of Chains↵
Khaleesi↵
Mother of Dragons↵
Unburnt↵
```

The following error may occur:

1. If the note's `note id` does not exist in the system, the error message is (Note <note id> does not exist!).

The following example illustrates these error messages.

```
tags Things Jon Snow knows↵
Note Things Jon Snow knows does not exist!↵
```

2.10 `tagged` command

Lists the notes tagged by this reference note. The command receives as argument the `reference note id` and prints a list of all notes tagged with this reference note. The listed notes are presented in case-sensitive alphabetical order of referenced notes id.

```
tagged ♯dead↵
Khal Drogo↵
Missandei↵
Ned Stark↵
Rob Stark↵
Robert Baratheon↵
Tywin Lannister↵
Viserys Targaryen↵
```

The following error may occur:

1. If the reference note's **`reference note id`** does not exist in the system, the error message is (Tag <reference note id> is not used!).

The following example illustrates this error message, when we try to list notes referenced by an unused reference note id.

```
tagged Dan and Dave improved this part of the story↵
Tag Dan and Dave improved this part of the story is not used!↵
```

2.11 `trending` command

Lists the most popular tags. The command receives no arguments. The tags (i.e. the corresponding reference notes ids) are listed in order of entering the most popular tags list, one by each line. If there are no reference notes yet, the output message should be (No tags defined yet!), but this should be regarded as normal behaviour (as opposed to an error situation).

```
trending↵
No tags defined yet!↵

... a few chapters later ...

trending↵
♯dead↵
♯don't get attached↵
```

The most popular tags in the system are those with the maximum current number of references from other notes. You need to be able to maintain these values as tags are added and removed from notes. As an example, consider that the currently most used tags in the system are **`♯dead`** and **`♯alive`**, both with 23 references. Now, suppose that the note **`Daenerys Targaryen`** has none of these tags. If we tag **`Daenerys Targaryen`** as **`♯alive`**, **`♯alive`** becomes the most popular tag, with 24 references. If we also tag her as **`♯dead`**, then both **`♯alive`** and **`♯dead`** will have 24 references. As **`♯alive`** reached this level earlier than **`♯dead`**, it shows up first in the output of `trending`. If we then untag **`Daenerys Targaryen`** as **`♯alive`**, **`♯dead`** becomes the sole most popular reference note.

```
trending↵
No tags defined yet!↵
trending↵
♯dead↵
♯alive↵
tag Daenerys Targaryen↵
♯alive↵
Daenerys Targaryen tagged with ♯alive.↵
trending↵
♯alive↵
tag Daenerys Targaryen↵
♯dead↵
Daenerys Targaryen tagged with ♯dead.↵
trending↵
♯alive↵
♯dead↵
untag Daenerys Targaryen↵
♯alive↵
Note Daenerys Targaryen no longer tagged with ♯alive.↵
trending↵
♯dead↵
```

2.12 `notes` command

Lists all notes of a given type last edited within a given time interval. This command receives as arguments the **`note kind`** (either **`permanent`** or **`literature`**), the starting date and the ending date. The list is sorted by date. Within the same date, the notes are sorted by order of last update (which in some cases can be the same date as creation, if the note was never updated) in the system. If there are no notes to list within the time frame defined by the start and ending dates, the message should be (No notes in this time frame.). This is considered normal (as opposed to an error situation).

```
notes permanent↵
2025 05 05↵
2025 05 06↵
Robert Baratheon marriage↵
Robert Baratheon brothers↵
Robert Baratheon↵
```

The following errors may occur:

1. If the note kind is neither **`permanent`** nor **`literature`** the error message is (Unknown note kind!).
2. If the starting date is invalid, the error message is (Invalid start date!).
3. If the ending date is invalid, the error message is (Invalid end date!).
4. If the ending date precedes the starting date, the error message is (The ending date must not precede the starting date!).

```
notes parent advisory↵
2025 05 07↵
2025 05 12↵
Unknown note kind!↵
notes literature↵
2025 05 37↵
2025 05 12↵
Invalid start date!↵
notes literature↵
2025 05 07↵
2025 05 32↵
Invalid end date!↵
notes literature↵
2025 05 07↵
2025 05 02↵
The ending date must not precede the starting date!↵
```

2.13 `delete` command

Deletes a note. The command receives a **`note id`** and deletes the corresponding note. In the example, the system deletes the note on **`Robert Baratheon's brothers`**. Deleting a note can be a fairly complex task. You have to go through all notes referencing the deleted note so you can remove the references to the note you are deleting. This includes references from all kinds of notes including permanent, literature, and reference notes.

```
delete Robert Baratheon's brothers↵
Note Robert Baratheon's brothers deleted.↵
```

The following error may occur:

1. If the note to delete does not exist, the error message is (Note <note id> does not exist!).

The following example illustrates a failed attempt to delete **`Ramsay Bolton's good deeds`**. Spoiler alert: there are none.

```
delete Ramsay Bolton's good deeds↵
Note Ramsay Bolton's good deeds does not exist!↵
```

3 Developing this project

Your program should take the best advantage of the elements taught in the Object-Oriented Programming course. You can use all the techniques discussed in the course lectures. In particular, this project will focus on the Java Collections discussed in class and the usage of the Java exceptions mechanism, so make sure you leverage these two elements in your solution. You should make this application as **extensible as possible** to make adding new kinds of notes easier. Use Java's LocalDate class to store dates.

You can start by developing the main user interface of your program, clearly identifying which commands your application should support, their inputs and outputs, and error conditions (hint: use the exception handling mechanism to cope with user input mistakes as described in this document). Then, you need to identify the entities required to implement this system. Carefully specify the **interfaces** and **classes** that you will need. You should document their conception and development using a class diagram, as well as document your code adequately, with Javadoc. You can and are supposed to use the Java Collections library presented in the lectures and labs.

It is a good idea to build a skeleton of your `Main` class, to handle data input and output, supporting the interaction with your program. In an early stage, your program will not really do much. Remember the **stable version rule**: do not try to do everything at the same time. Build your program incrementally, and test the small increments as you build the new functionalities in your new system. If necessary, create small testing programs to test your classes and interfaces.

Have a careful look at the test files, when they become available. You should start with a bare-bones system with the `help` and `exit` commands, which are good enough for checking whether your command interpreter is working well, to begin with. Then, add permanent notes, still with no links. List them. All good? Implement the update command for those permanent notes. If the listings are still ok, move on to literature notes and do the same. After you have basic permanent and literature notes, create links among these notes. Make sure you can list them. When you can, develop the link removal feature. Move on to the tags functionalities, then the listings, and, finally, the note deletion commands. The order in which the commands are presented in this document provides you with a reasonable sequence for your implementation, which is also in line with the order of testing suggested by the tests that will be made available to you.

Step by step, you will incrementally add functionalities to your program and test them. **Do not try to develop all functionalities simultaneously. It is a really bad idea.**

Last, but not least, **do not underestimate the effort for this project. START EARLY**.

4 Submission to Mooshak

To submit your project to Mooshak, please register your group in the Mooshak contest POO2025-TP2. The mooshak login for the group must be the concatenation of the numbers of the students that make up the group, separated by _ (the first number must be the smallest). For example, students #5678 and #5677 should have their Mooshak username **5677_5678**. **Only the projects submitted for evaluation through logins following the above rule will be accepted.** The name and number of students that make up the group must be inserted in the header of all submitted files, as follows:

```
/**
* @author STUDENT1NAME (STUDENT1NUMBER) STUDENT1temail
* @author STUDENT2NAME (STUDENT2NUMBER) STUDENT2temail
*/
```

4.1 Command syntax

For each command, the program will only produce one output. The error conditions of each command have to be checked in the same order as described in this document. If one of those conditions occurs, you do not need to check for the others, as you only present the feedback message corresponding to the first failing condition. However, the program does need to consume all the remaining input parameters, even if they are to be discarded.

4.2 Tests

We expect you to create your tests based on this specification. Create tests for the “happy day scenario” and error situations so that whenever you add support for a new command, you also create suitable tests. So, build your tests incrementally as you make your project. This mimics what happens with the Mooshak tests. The Mooshak tests verify incrementally the implementation of the commands. They will be made publicly available by May 19th, 2025. When the sample test files become available, use them to test what you have already implemented, fix it if necessary, and start submitting your partial project to Mooshak. Do it **from the start**, even if you just implemented the exit and help commands. By then, you will probably have much more than those to test. Good luck!