Introduction to Object-Oriented Programming: Project

Practical Guidelines

- Students work on the project **individually!** Code duplication between students will result in severe penalties for all parties involved; no code copying from online sources allowed.

- Questions regarding the project assignment should be asked in **the Toledo forum (Assignment Questions)**. Only these questions will be answered, **no questions will be answered via e-mail**; we will not help with technical questions (i.e., fixing bugs).

- The deadline for the final project submission is **Friday, January 6th 2023, 23:59**. **No projects received after this date will be accepted! The deadline will not be extended, no exceptions!**

- An electronic version of your project should be submitted via Toledo, via the Assignments module. As discussed during the introductory lecture, multiple submissions will be allowed. However, Toledo should not serve as your daily backup service, and hence **the number of submissions allowed will be restricted to 5! No exceptions** will be made (regardless of any reasons you might have); the idea is that you upload your project code when it's complete and fully functional, but you can still submit corrections in case you find a bug or another type of problem. Projects that are late will be discarded; **the final submission will be used for grading** (again, **no exceptions!**).

- **Only submissions via Toledo will be graded, submissions via e-mail will be discarded! No exceptions, so submit your assignment on time!**

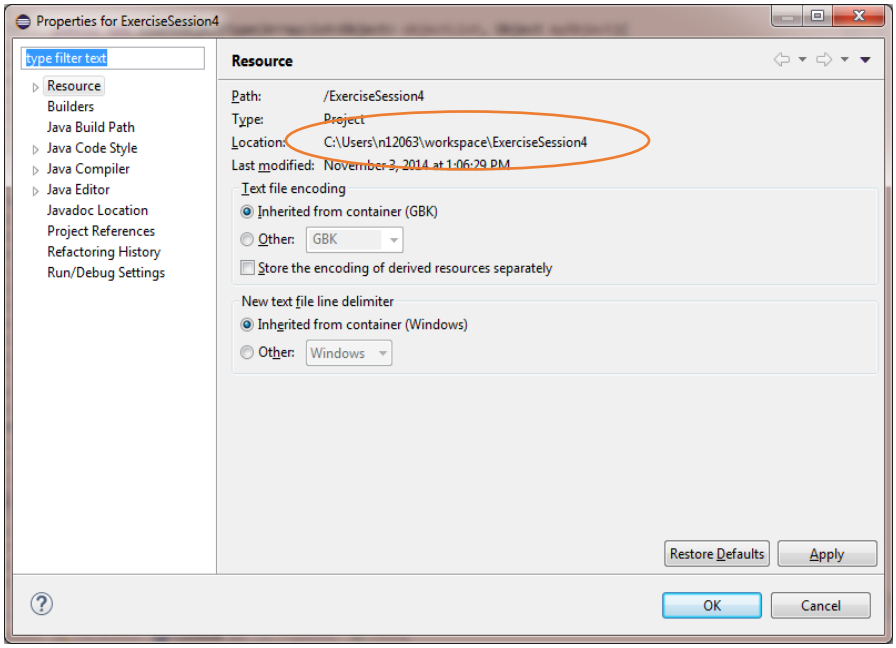
- The Toledo submission system will indicate a mark of 10 as the system requires a maximum mark to be provided. This is not relevant, and the details on how grading is done for the course is explained in the introductory lecture / slides.

- You must include the following components in your project submission, preferably in a zipped file with **surname and first name** as the name of the file: **surname_firstname.zip:**
  - Java project source code copied from your Eclipse / IntelliJ workspace (i.e., the src folder), including all the required files (when applicable) for your project to run in the correct location; no .class files or .jar files should be submitted, do not include the /bin directory; your project should contain exactly **one main method**.
  - Short report describing your project (**maximum** 4 pages, see guidelines) with strong focus on the structure and design of your code; no UML diagrams should be provided.

- There is no (oral) defense for this project, so invest sufficient time and effort in your report.

- Prepare your report and code (names, comments, etc.) in **English**.

- To copy your source code from Eclipse:
  - From Eclipse, right click on your project, select Properties. Under Resource, you will find the location of the project folder.
  - Navigate to this location on your computer and include a copy of the src folder in your project in your submission.



- To copy your project from IntelliJ:
  - In IntelliJ, you can see the location of your project in the Project view/panel (next to the project name that's listed in bold).
- It's a good test to see if you can copy your project to another computer and see if you can run it there from within an IDE such as Eclipe, IntelliJ or Netbeans. Most programming environments have the possibility of "Creating a Project from Existing Local Sources".
- Another good test is to try to run your application on a different operating system (Mac OS, Linux, Windows, …) than the one you used during the development process.

Coding Guidelines

The code guidelines below are a selection from a more extensive set of guidelines provided by Oracle (and can be found on the oracle website). We expect you to follow at least these guidelines:

- Use a separate file for each class and interface
- Assign classes that are not related to each other to different packages
- Write at most 1 statement per line
  ```
  int x = 5;       //(good)
  int y = 10;
  ```
  is preferred over
  ```
  int x = 5; int y = 10;       //(bad)
  ```
- Naming conventions
  - **Classes**
    - Class names should be nouns, in mixed case with the first letter of each word capitalized. Try to keep your class names simple and descriptive. Use whole words—avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form).
      - Start with upper case, e.g.: `Person`, `Car`, `Game`, `BMICalculator`
  - **Interfaces**
    - Start with capital letter, e.g.: `Nameable`, `Capable`, `Pettable`
  - **Methods**
    - Names are usually verbs, and should start with lower case and use camelCase if it consists of multiple words, e.g.: `getValue()`, `add()`, `performSomeAction()`
  - **Variables**
    - Variable names should be short (where possible) yet **meaningful**!. Choose variable names which indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are: `i`, `j`, `k`, `m`, and `n` (for `int`) and `c`, `d`, and `e` (for `char`).
    - Name starts with lower case and use camelCase if it consists of multiple words, e.g.: `name`, `dateOfBirth`, `ageLimit`, `stomachContent`
  - **Constants**
    - Should be all uppercase, use "_" to separate words, e.g.: `PI`, `DAYS_IN_WEEK`, `GRADE_TO_PASS`
- Provide comments for each variable, method, and class so that it is 100% clear to the reader of your code what its purpose is.
- Use proper indentation for your code. Every time you start a block of code (when making a class/method/if-else if-else/switch/…) indent the code within the block.
  - TIP: Eclipse and IntelliJ can do this for you.
    Eclipse: Select the code you want to indent properly > right-click > source > Correct Indentation
    IntelliJ: Select the code, Code > Auto-Indent Lines

Project Guidelines

Description of the FASTA (.fasta) input file

For this project, you need to create a command line implementation of a bioinformatics team that works together on a nucleotide multiple sequence alignment (MSA). There is no need to familiarize yourself with the specifics of how such an alignment is constructed or what it can be used for. In other words, the information provided in this document should be sufficient. **Two example input files** are provided on Toledo.

Whenever someone becomes infected with a pathogen, such as the currently circulating corona or monkeypox viruses, bioinformatics tools can be used to study the genome sequences of that pathogen. To this end, the genome sequences (consisting of 4 possible nucleotides / characters: A, C, G and T) of a collection of samples are investigated by first creating a multiple sequence alignment (essentially, putting the same positions within the virus' genomes below one another). The .fasta input file on Toledo contains such an alignment (for HIV), in a commonly used file format known as FASTA. Basically, for each sequenced sample, the FASTA file contains 2 lines: first, the identifier (or name) of the sample (the lines that start with a '>'), followed by the corresponding genome sequence. A small snippet (i.e., part of the first 3 genomes in the provided file) of the provided (standard) alignment can be seen here, taken from the provided file on Toledo:

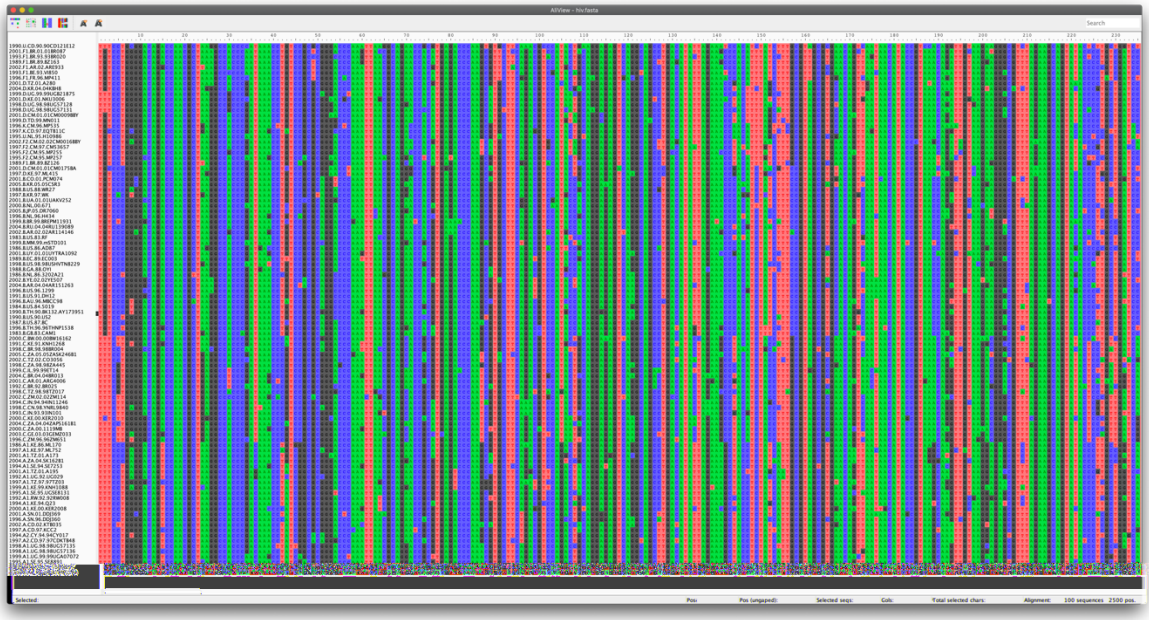
>1990.U.CD.90.90CD121E12
TTTCCTGCGGACAGACCAACGCTAAGGCCACCCC …

>2001.F1.BR.01.01BR087
TGTCCTGGGGACAGACCAACGCTAAAGCCACCCA …

>1993.F1.BR.93.93BR020
TGTCCTGGGGACAGACCAACGCTAAAGCCACCCA …

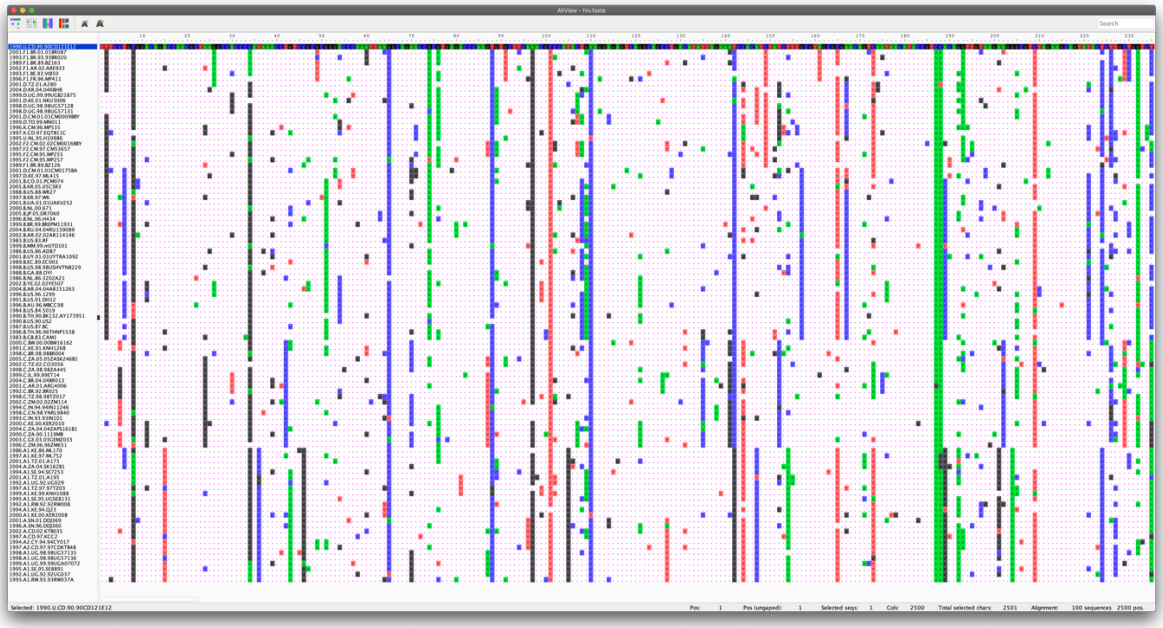
As each genome in a FASTA file is characterised by 2 lines (the name / identifier of the genome, preceded by a '>' symbol and on the subsequent line the actual genome, only shown in part above), the .fasta file on Toledo contains 100 genomes – each of length 2500 nucleotides (i.e., the characters A, C, G or T) – and hence consists of 200 lines. Your implementation should be able to handle FASTA files of different dimensions, i.e., with a different number of genomes and also with genomes (or the overall alignment) that are of different length. You can assume that for this assignment, no other characters will be part of the provided genomes.

If you are interested in exploring the alignment as a whole (not mandatory), you can for example use the freely available software package AliView: https://ormbunkar.se/aliview/ A screenshot of what you should see on screen is provided below. You'll notice that each genome occupies one row in AliView: in the left part of the visualisation, you can see all the names / identifiers, and on the right (in color), you can see the genomes. Each nucleotide / character is given its own unique color so that differences between the genomes are easily visible.



For the purpose of this assignment, you need to implement 2 types of alignments: the standard alignment as discussed above, and what we will refer to as a SNiP or SNP alignment. An SNP alignment is a different way of representing the same data as in the standard alignment, i.e., no separate input file is required from which to read a SNP alignment as you can obtain it directly from the standard alignment. Compared on a reference sequence, which is shown in full (see screenshot below) as in the standard alignment, only the (nucleotide) differences in the other genomes are shown, while identical nucleotides are represented as a dot ('.').

You can visualise such a SNiP alignment in AliView by following these steps: select a genome (typically, the first / top genome is used as the reference sequence) and select View > Highlight diff from a selected sequence. You should then see the following visualisation:



Each alignment type needs to support at least the following operations:

- search through the genomes for a specific sequence of characters (e.g., AACAAATG) and return the corresponding names / identifiers for those genomes in which the sequence can be found
- replace a genome in the alignment with a new sequence (i.e., a genome that was previously not part of the alignment and that has a name / identifier that was not part of the initial FASTA file)
- in a given genome, replace all occurrences of a given sequence of characters by a new sequence of characters (without changing the total length of the genome)
- in the entire alignment, replace all occurrences of a given sequence of characters by a new sequence of characters (without changing the total length of the alignment)
- adding a genome with its corresponding name / identifier
- removing a genome, based on its name / identifier
- other functions that you think are useful to edit an alignment

It will be **important** to think about what effect some of these methods will have on the different alignment implementations. Finally, and to be 100% clear: you do **NOT** need to develop a graphical interface / visualisation of the alignment(s) as part of this assignment. The images provided above (from AliView) only serve to explain the data you will be working with.

Description of the text (.txt) input file

A bioinformatics team consists of a potentially large number of users that each have their own responsibilities and possible tasks they can perform. An example input text (.txt) file is provided on Toledo, describing each team member by her/his function, first and last name , and years of experience in the team. You can assume that a team member has a first and last name that consists of only one single word (without special characters).

The goal of your application is to construct a standard alignment for each bioinformatician to work on, based on the data in the provided .fasta file. **Both the .fasta and the .txt file can only be read in once!** The .fasta file also serves as the input to create the optimal alignment that is stored in a repository, which is initially identical to each of the bioinformaticians' alignments. The repository also contains the SNiP alignment that **always** corresponds to the current version of the optimal alignment. Only the team lead(ers) and the technical support members have access to the repository. The team lead(ers) and technical support can query the repository for each user's personal alignment, but they cannot retrieve (nor directly access) the optimal alignment nor the SNiP alignment from the repository.

As you have been able to deduce from the previous paragraphs, there are three types of team members:

- bioinformaticians, who only have access to their own personal alignment (that can change over time due to editing operations), but not to the repository

- team lead(ers), who have direct access to a repository that contains the optimal alignment (with the lowest score), its corresponding SNiP alignment and the personal alignment of each bioinformatician, i.e., the team lead(ers) do not have their own alignment to work on but can select the alignment of any user to be promoted as the optimal alignment based on a simply 'difference score' criterion; the team lead(ers) are not able to retrieve the optimal alignment nor the corresponding SNiP alignment from the repository, but the alignment from each specific user can be retrieved

- technical support, who have direct access to the same repository as the team lead(ers); however, the functions of technical support members restrict themselves to backing up, restoring and removing / clearing the data in the repository

The following functions / operations need to be provided for the bioinformaticians and the team lead(ers), but not for the technical support crew:

- write data to file: for bioinformaticians, this means writing their own personal alignment to an output text file, of which the name is simply their first name + their last name with a .alignment.txt extension; for team lead(ers), this means writing all of the users' alignments to one single file with the same naming convention
- write a report to file: for bioinformaticians, this means writing the difference score (see below for more information) for their own personal alignment to an output text file, of which the name is simply their first name + their last name with a .score.txt extension; for team lead(ers), this means writing all of the users' alignments scores to a file with the same naming convention

The 'difference score' of an alignment is a simple metric to compute. Basically, you use one of the genomes in the alignment as the reference genome (typically the first / top genome) and compute the number of different positions / characters of each other genome compared to that reference genome. Add up all of those numbers to obtain the difference score. Importantly, this difference score also needs to be defined for the SNiP alignment! You can – for example – have this score computed in your main method to show how you have implemented this aspect.

The following functions / operations need to be provided specifically for the bioinformaticians:

- the functions that enable changes to the alignment, as listed above
- retrieving the personal alignment (from that bioinformatician)

The following functions / operations need to be provided specifically for the team lead(ers):

- copy a user's alignment to the optimal alignment in the repository (note that this will also affect the shared SNiP alignment); after this operation has been performed, changes to the user's alignment do not affect the optimal alignment (nor the other way around)
- overwrite a user's alignment with the optimal alignment; after this operation has been performed, changes to the user's alignment do not affect the optimal alignment (nor the other way around)

The following functions / operations need to be provided for the technical support members:

- backup the repository data: a hard / deep copy of the current optimal alignment, its corresponding SNiP alignment and, for each user, her/ his personal alignment; whenever such a backup is made, the date and time of the backup procedure is stored as an instance variable for the technical support member
- restore the repository data: reinstating the backup data, and hence overwriting the contents of the current optimal alignment, its corresponding SNiP alignment and, for each user, her/ his personal alignment
- clearing the repository data: removing / emptying the current optimal standard alignment, its corresponding SNiP alignment and, for each user, her/ his personal alignment

The goal of the application is to provide the opportunity for each bioinformatician to work on their own personal alignment independently, through the operations mentioned earlier in this document. As such, each bioinformatician starts off with a personal copy of the same initial standard alignment, which she/he can independently work on and make changes to. As different bioinformaticians perform different operations on their personal alignment, they can hence end up with different alignments during the course of their work. The initial alignment also serves as the initial optimal alignment in the repository and as a source for creating the initial SNiP alignment. None of the users, regardless of their function, is able to edit the optimal alignment(s) directly, as the optimal alignment can only be copied as a whole, as described above in the functions for the team lead(ers).

Description of the application / main method:

First of all: provide sufficient comments in the code, especially in your main method, but also throughout the other classes you decide to implement.

In your main method, the goal is to show how the different classes you decide to implement work together, and also what their main features are and how to use them. For example, start by reading in the two input files, constructing the required alignments (and printing them after having done so) and users, and have each user perform a number of tasks as described in this document. Then have the team lead(ers) and technical support personnel come in and perform a few tasks (as described above), after which the bioinformaticians can continue working on their alignments. A short (incomplete) example of what the output could look like is shown here (but yours should be more extensive / elaborate):

…

SNiP alignment:

>1992.A1.UG.92.UG029
....C..A.....G.....C.....C.....A.....T.....G.G..... (full line not shown)
>1998.A1.UG.98.98UG57135
....C.....T...T.....A.....G.....G.....A.. (full line not shown)

>1998.A1.UG.98.98UG57136
....C.....C.....A.....G.....G.....A... (full line not shown)

alignment score = 34443

SNiP alignment score = 34443

Replacing TTTTC with TTTTT in genome 1998.A1.UG.98.98UG57136
Marc Janssens's alignment score = 34444

Promoting alignment from Marc Janssens to shared alignment

Copying shared alignment to Werner Lippens

Werner Lippens 's alignment score = 34444

…

Providing input to your implementation:

At this point, the only aspect left to discuss for your application is how to instruct your implementation which text and FASTA file to read. Hard coding file names into your application **should be avoided**, and there are various options available to achieve this. The easiest one would be to provide this information via the program / command-line arguments, but you can also make use of a properties file. Which of these approaches to implement is entirely up to you and does not affect your score in any way. You may assume the presence (and name / location) of such a properties file in your implementation if you so choose. More information can be found here (for example):

https://www.jetbrains.com/help/idea/properties-files.html

https://stackoverflow.com/questions/30010833/creating-a-properties-file-in-java-and-eclipse/30010882

For this assignment and the specific files discussed in this document, such a properties file (typically named config.properties) only needs to contain two lines of key-value pairs:

teamfilename=team.txt
fastafilename=hiv.fasta

Important: both input files should only be read in **once** at the start of your program / implementation. In your main program, show how to use the functionality you have implemented in the different classes of your implementation and how the different classes are to be used together.

Important: make sure that you use relative paths to the file that needs to be read so that your application works directly when copying your Eclipse / IntelliJ project files. Hard coding the location of the file into your implementation is considered poor practice.

Key points when implementing the assignment:

The structure of your code is the most important evaluation for the project. This means you should take care to include as many of the **object-oriented concepts covered in the course** whenever applicable. That also means thinking about future extensions of your implementation and keeping an eye on possible code reuse (outside of the current assignment).

Watch out for plagiarism! Online you may find similar partial implementations which you may use for inspiration, but you must write your own code! Note that many code examples you can find online are poorly written and/or poorly designed.

We realize that design patterns such as Model-View-Controller are not part of the course material, and you are **not** meant to study this material for the project; you are however required to think about properly structuring your project code.

Do not implement your project with solely the provided input files and their dimensions / number of lines in mind. Your project should run fine with a different .fasta and .txt files as well. Avoid hard coding aspects that relate to the specifics of these files at all cost.

Finally, aim for a well-designed / well-structured project with cleanly written code, **organized in different packages**. As stated before, clearly state in the project if you were unable to provide certain requested aims of the project and provide some information (what did you try and where did it go wrong).

Questions can be asked on the Discussion Board in Toledo.

Report Guidelines

You should prepare a short report of **maximum 4 pages** detailing important components of your project. It should include the following information:

- Write a short description of every class, indicating what functionality is included in each class.
- Describe the relationship between your classes (this may be text and/or a simple diagram, but not a UML diagram), for example, inheritance relationships or method calls from one class to another.
- If you are unable to implement a certain part of the game, we encourage you to make a sensible decision, be upfront about this decision (i.e., explain it in your report) and explain what the main difficulty was. Additionally, if you encounter implementation difficulties or time constraints, it's better to fully and correctly implement a subset of the functionality rather than to implement small parts of each of the targeted program parts.
- Discuss what you think are the **strengths and weaknesses in your project** but focus on code design when doing so and not on how the application looks. Describe any difficulties you faced while working on the project.