Assignment 1

This first assignment is a not-so-gentle introduction to the Java language and the Eclipse IDE that we will be using. You will accomplish a number of tasks in this assignment:

- You will create a GitLab[1] account to hold your solutions repository.
- You will create a *fork* of the assignment repository from my GitLab account to yours.
- You will use GitKraken[2] to *clone* this repository to your local computer.
- You will use Eclipse[3] to review and edit the Java code for the assignment, and to compile the project files into executables.
- You will use the Gradle[4] tool to manage your Java project, run tests and check your code style.
- You will get up to speed with the basics of the Java language by implementing a double-linked list and studying the basic building blocks of Java Object-Oriented Programming.

Let's get started!

Setting up GitLab

GitLab is a site that maintains code repositories. If you are not familiar with it, version control systems are systems that allow us to maintain the history of evolution of our code over time, to identify where and when problems occurred and how they were fixed etc.

There are three tasks on this section:

- Create a GitLab account if you don't already have one
- Find and fork my repository so you have a copy of it on your account
- Allow me access to your repository, while restricting access to others.

If you do not have a GitLab account already, the first job would be to create one. Go to www.gitlab.com to create an account (use the **Sign Up** option on the top right). It will likely ask you to confirm the account from your email, so make sure to check your email after signing up.

[1] https://gitlab.com/
[2] https://www.gitkraken.com/
[3] https://www.eclipse.org/
[4] https://gradle.org/

> GitLab and GitHub accounts are an important part of your public persona as a programmer. Prospective employers may look at your projects there. Choose a suitable name, and put a good picture up in your profile.

The next step we need to carry out is for you to *fork* the assignment repository, which resides on my account. A fork of a repository allows for independent development of that project. This way you can add your solutions to your repository without messing with my repository.

Now you will need to find my repository. Just go to this address[5] and you should be starting at the main page of a project called algorithms–assignments. You should be able to see a **Fork** button. Click on it and your own version of this project will be created. You will now be staring at a similar page, but over at your account (your login name should appear prominently).

Now that you have the repository forked, we must properly set it up. There are two things to ensure: That others can't access it, and that your instructor can access it.

In order to do that, we need to go to the project **Settings**. This should be a gear button to the bottom left of the screen. Start with the **General** section, and its **Permissions** part. Make sure that the **Project visibility** is set to **Private**, then **Save Changes**.

Next go to the **Members** section of the project **Settings**. There type skiadas to find and select my account, then choose the role permission to be **Reporter**. Finally, click **Add to project**.

Now you are set! We will move on to GitKraken

Setting up GitKraken

GitKraken[6] is a desktop program to manage your repositories. When we have some code that we want to *commit* as done, we will use GitKraken. Here are the main topics we will cover in this section:

- Start GitKraken and synchronize it with your GitLab account
- Clone your repository to your local computer
- Familiarize yourself with the interface
- Make a change to a file and make commit

Let's get started! You will find GitKraken in the Developer section. You will need to create a GitKraken account to really work with it. You do NOT need the "Pro" account.

Now you will see the main GitKraken window. Start by setting up your preferences at the top right. You may need to edit your **profile** to include a name and email, and these

[5] https://gitlab.com/skiadas/algorithms-assignments
[6] www.gitkraken.com

need to match those your provided to GitLab. Then go to the **Authentication** tab, select the **GitLab.com** tab, then click to **Connect to GitLab**. Your browser window should open, and you should **Authorize** the shown request. You can then close the browser window and go back to GitKraken. There it may show you a green button to **Generate SSH key and add to GitLab**. Do that. Then exit the preferences at the top left.

Next we will need to find your project. Click the folder at the top left, and choose the **Clone** tab, then the **GitLab.com** tab. You should now see your repository there, select it. Then in the **Where to clone to** area you should use the **Browse** button to find a good parent folder for your project. I suggest that you create a new folder for this class, and select that. If all has gone well, you should see an **Open Now** option at the top of the GitKraken window.

You are now starting at the main GitKraken window with a repository open. You are likely seeing a list of commits by me in the middle. They all contain a brief message, and you can click on one of them to focus on it; currently the newest one, at the top, is focused. On the right you see what files this commit has changed. You can click on one of those files to see what changes have happened, and then the main view changes to the **file differences* view, which shows you the differences introduced by this commit. The red lines indicate text that was there before but is now removed, while green lines indicate text that is being added.

We will start by creating your first commit. We will use SublimeText for this, even though later we will do most of the project work using Eclipse. For now, open a terminal and `cd` your way into the `algorithms-assignments` folder contained in the folder you created earlier. Then type `subl .`, and it should open up the entire project in SublimeText. Alternatively, you can fire up SublimeText and then choose the **Open Folder** option from the **File** menu (though learning to use the terminal is an important skill to learn).

Now, in SublimeText, look into the `src/main/java/utils` folder path, and open the `DLList.java` file that is located there. We will look at this file in detail later, but for now find the `@author` line and change it from my name to yours, then save. Then go back to GitKraken, and you should now see a new and different item in the middle screen, titled WIP for “Work in Progress.” This shows you changes that you have made but not yet committed. Click on the circle to choose it, then click on the file on the right to see the changes.

We now need to commit these changes. This is a two-step process: We **stage** which changes we want, then we make a **commit**. Hover over the filename on the right, and you should see a **Stage File** button. Click it, and you will see the file moved to the “staged files” area. When you have made multiple changes, you can use this system to only commit some of them.

Next you need to do some work in the **Commit** area at the bottom right. Add a “summary” of your commit, something along the lines of “Change the author name”. Then click the **Commit changes** button at the bottom. Congratulations, you have just created your first commit!

It still only lives locally in your computer. You can see that by noticing that you now have two *master* tags in the middle window. The one has a little computer icon in it, indicating that it is your *local* copy. You should see it one step ahead of the other one, which is the *remote*. When you are ready to upload your changes to the GitLab site, use the **Push** button at the top of the window. You will then see the two tags synchronize.

Take a deep breath, this was a lot of new stuff. Take a moment and visit the GitLab site, and see the changes you made to your project show up there.

Next we will take a look at **Eclipse**, a specialized IDE with many convenient facilities for Java programming.

Setting up Eclipse

In this section we will learn to use Eclipse, and *Integrated Development Environment* (IDE) for Java and other languages. IDEs offer a lot of useful functionality for larger programs.

In this section we will learn the following:

- Start Eclipse and open our project in it
- Set up and use Gradle tasks to perform common operations
- Build the documentation for the project
- Run the project's tests and see the results

Let's get started! You will find the Eclipse app in the Developer menu. Start it. The first thing you need to do is select a directory for workspace, just let it choose the default directory. After a while the Eclipse welcome window will show up. Choose the option **Checkout projects from Git**, then **Existing Local Repository**, then use the **Add* button and** Browser** to find your cloned repository, then select it and Finish. This will add the repository to the link of repositories that Eclipse knows about. Next time around you won't have to go through this process again.

Now we need to open that repository. Select it and click Next. You should have an option to **Import existing Eclipse project**, and the **Working Tree** should be selected. Click Next, and finally Finish one more time. You should now be starting at the main Eclipse window. There are some errors at the bottom that we will get to in a minute.

NOTE: If you exit Eclipse, the project will likely be open for you the next time you start Eclipse again. Just close the Welcome screen and you should see it.

Now, let us take a look at a file in Eclipse. Navigate into the src/main/java/utils folder, and you will find a DLList.java file. Double-click it to make it show up in the main view. Take a moment to familiarize yourself with this view:

On the left side of the code you will see some red markers, identifying problem areas (we will fix them shortly). Mousing over one of them will show you more details of the error. You also will see some red rectangles on the right side. Those point to other locations in the file where problems occur, and you can click on one to go there. There are also some blue rectangles. Those point to TODO sections, and you will need to later fill those in with your code in order to complete the assignment.

Let us resolve some of the problems we are having. The main problem is that we have to tell Eclipse about the Java build path. Make sure the top project element is highlighted on the left, and go to the **Project -> Properties** section. There are a lot of options you can set up here, but the one we are interested in is the **Java Build Path** section. This is where we specify to Eclipse where to look for relevant Java files. Select it, then go to the **Libraries** tab, and **Add Library** on the right. Then use the **JRE System Library**, and choose the **Workspace default JRE**, then **Finish**. Next go to the **Order and Export** section, and click on the new java-8 item there. Then click on **Apply and Close**. You should now see most of the error markers from the DLList.java file go away.

Next we need to make sure that Gradle is set up properly. Gradle is a software that adds some convenient tasks to Eclipse. Start by going to the **Help** menu and its **Eclipse Marketplace** section. Search for **Buildship** in the searchbox, and after a few seconds the **Buildship Gradle Integration** entry should show up. It should also say "installed" at the bottom right. Go ahead and close the window in that case.

We now need to take a step from a terminal window. You can open up a Terminal window within Eclipse by typing **Ctrl+Alt+T**. Do that now and you should see a small terminal View show up at the bottom. In it we need to run the command:

```
sh gradlew cleaneclipse
sh gradlew eclipse
```

Each of these will take a while as it installs some items. Once this is successful, right-click the top project item on the top left, and choose **Refresh** from the context menu. Then right-click it again, and under **Configure** you should see an **Add Gradle Nature** option, click it.

We next want to bring the **Gradle Tasks** view up. Look under **Window -> Show View -> Other**, and inside the **Gradle** section you will find **Gradle Tasks**. Select it and you should see a new Gradle Tasks tab at the bottom. This bottom section contains what are called *Views*. These are various windows with information about the project. We already saw the Markers view, the Terminal view, and now the Gradle Tasks view. You can always add more views from the **Window -> Show View** menu.

Now look at the **Markers** view at the bottom left, and you should no longer see any red items. There are some Java Problems and Java Task items, feel free to look at them but do not worry about them too much right now. The Java Task section lists all the places that you need to add code to complete the assignment, so you may use it later on.

For now, switch to the **Gradle Tasks** view. If it looks like it has no items in it, then close the view and open it again from the **Window -> Show View -> Other** menu. You should be seeing an *algorithms-assignments* element that can be expanded. While you can view the tasks this way, I find it easier to not group them that way. In order to change this, find the little downward arrow near the right of the view window. This brings up view settings. Uncheck the **Group Tasks** setting.

These are various tasks related to our project. You can execute one of these tasks by simply double-clicking it. Do this now for the **build** task, which basically compiles all project files. You should see some errors near the bottom, as the various unit tests are failing since you have not implemented your classes yet.

Let's try another Gradle task. It is called **javadoc** and it generates documentation for the project based on the comments. Go ahead and run it. This created documentation, but we need to find it. It is contained in a build folder that by default is not showin in Eclipse. To make it appear, use the little downwards around on the left panel, go to **Filters and Customization**, and de-select the **Gradle Build Folder**. Then Click **OK**, right-click at the top project and choose **Refresh**, and you should now see a **build** folder on the left side. In it you will find a docs folder, and in it a javadoc folder. Right-click on the index.html file and choose **Open With -> Web Browser** to open up the documentation.

Good, you are all set and ready to start the assignment! Take a short break, you've earned it. Next we will discuss some key Java elements, before looking at the details of the assignment itself.

Object-Oriented Programming and Java

In order to follow along, make sure to have open the DLList.java file in Eclipse. You will find it within the src/main/java/utils path. It is part of the utils package we are creating.

You are already familiar with classes from C++. In Java **everything** is in a/part of a class. In particular you cannot have any stand-alone functions, all functions must be part of a class, either as **object methods** or as **class methods**.

The DLList class is a class that models a double-linked list, which uses a *sentinel/-guard node* to mark the beginning and end of the list, resulting in effect in a circular list. Each node contains an element, as well as references to the previous and next nodes. The **front/start** of the list is the node that is *next* of the sentinel, while the **back/end** of the list is the node that is *previous* to the sentinel.

The class definition starts around line 26, following a large comment using the Javadoc conventions of documentation. It starts with a line like so:

```
public class DLList<E> extends AbstractSequentialList<E> implements Deque<E> {
    ...
}
```

There is a lot of information there, so let us take it one step at a time.

- The keyword class indicates that this is a class, while the word public means that this class is visible to all.
- The name of the class is DLList<E>, meaning it takes as a parameter another class E (the class for the elements contained in our list).

- The class extends AbstractSequentialList<E>. This means that our list is a *subclass* of the AbstractSequentialList class[7], whose page you should take a look at. Objects of a subclass must have at least the properties and methods that the original class had, and they may add their own properties and methods, or they may override some of the methods with their own implementations.

- The class automatically inherits the implementations of methods from its super-class, unless chooses to *override* those implementations. This is called **inheritance**. In this instance we inherit many methods from the AbstractSequentialList class, which you can see in the *Concrete Methods* section of the documentation. In order to benefit from all those we must provide an implementation for the *abstract methods*, which were not provided by the superclass. In this instance we have to provide implementations for listIterator and size, which we do later in the file.

- Lastly, the class also implements Deque<E>. Deque<E>[8] is an **interface**, whose documentation you should look at. Interfaces are a bit like the .h files in C++. They specify a certain set of methods, with specific *method signatures* in terms of the types of the parameters and the results. If a class wants to claim to implement an interface, it must provide all these methods that were advertised in the interface.

- Interfaces allow us to set boundaries between our classes. A class can work with objects from another class if it knows an interface that this other class is implementing. And the actual class of those objects can change, and as long as the interface remains the same then our code will work. This *separation of implementation and interface* is an essential component of proper OOP.

Within this class we may include various elements:

- We can have what are known as **object fields/properties**. Each object of the class will have values for those fields. In this case we have such an element, the *sentinel*, which is specified by the line:

  ```
  private Node sentinel;
  ```

  The private keywords means that entities outside the class cannot directly access this sentinel property. We will provide access to it if/as needed via *accessor* and *mutator* methods. In this case we actually will not, since the sentinel node is just an *implementation detail*, and entities outside the class need not even know that we have it.

  Then we specify the Node type. Node is a class we define later in the file, and we will discuss it a bit further. Finally, the name sentinel is the name we give to this field, so we can refer to it elsewhere in the class. Within methods of the class these fields can be accessed either directly with their name, or as this.sentinel, using the keyword this which refers to the object itself. We encourage you to always use this latter form, to avoid confusing *local* variables with these fields.

[7]https://docs.oracle.com/javase/8/docs/api/java/util/AbstractSequentialList.html
[8]https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html

- We can also have whole classes, so called **inner classes**. You can think of them as *helper classes* that the main class needs to do its work, and the rest of the world does not need to know anything about them.

  In this case we use three such classes, a Node class to manage the nodes of our list, and a DLListIterator class that we need to provide a list iterator. This would allow users to process elements in our list via a **foreach loop**, avoiding the need for array indexes. We also need another class to provide a "reverse list iterator" for traversing the list from the end. We will discuss these inner classes a bit in a moment.

- We then have **class constructors**. These are methods used to create the objects of the class. They are always named with the same name as the class, and we can have multiple constructors depending on what arguments each constructor takes. We only use one constructor for our class, found around line 253. It looks like this:

  ```
  public DLList() {
      this.sentinel = new Node(null);
      this.sentinel.next = this.sentinel;
      this.sentinel.prev = this.sentinel;
  }
  ```

  The keyword public means that others can call this constructor, which is good because otherwise noone would be able to create lists. The contents perform three assignments. the first sets the sentinel field to a new empty node. The other sets the previous and next fields of that node to point to itself. This is how the initial arrangement of the sentinel node would work. The fact that it points to itself means the list is empty.

- Finally, we have the various **methods** of the class. These may be private or public, depending on whether it is intended for external use or not. For example we have:

  ```
  private Node getNodeAtIndex(int index) {
      ...
  }
  ```

  which is a private method, which *returns* an object of class Node. It is named getNodeAtIndex, and it takes exactly one parameter, called index, of type int. We can have two methods with the same name, as long as they have different parameters or parameter types.

  Most of these methods have unfinished implementations. Your assignment would be to provide those implementations.

Before proceeding to your assignment, let us look at the Node class. Its implementation starts around line 177. You should study this code both as another example of a class and also to understand the key objects that you will be manipulating in your assignment. As you look through the code you should identify the following:

- The objects of this class have three fields to them, named item, prev and next. These are marked as public, so that the DLList class can make direct use of them.

- There are two constructors. One takes a single argument, the item to be stored, the other takes three arguments for the item as well as the previous and next nodes.
- There are three methods, addAfter, addBefore and removeSelf. Read the documentation preceding them as well as their implementation, to understand how they work. Draw some pictures to help you further. They are the key methods used for adding nodes to the list and removing nodes from the list.

The assignment

In the assignment you have to implement a number of methods for the class, related to the Deque interface. You can read about double-ended queues here[9]. In important thing is that these can be used to provide a *stack* interface as well as a *queue* interface, and both of these will be useful later on. You should be already familiar with these basic concepts. We have already implemented the stack and queue methods on top of the deque methods at the end of the DLList.java file.

You will find numerous TODO comments through the file. They mark the locations where you need to implement a method. You should read the documentation of the java Deque interface[10] for what these methods are supposed to do, though we will briefly discuss it here. If you use Eclipse, you can see blue rectangles on the right side of the code indicating the location of all TODOs, and you can click on them to go there. You can keep track of your progress that way, so make sure to remove the TODO from the methods you have already implemented.

This is not a requirement, but you should also add your **own tests** to the /src/test/java/DLListTest.java class, or you can create your own class there. Some start tests have been provided for you. You can use the :test Gradle task to run the tests.

You should make a new *commit* in GitKraken after each method you write, do not try to commit everything at once.

When you are done with the assignment, in order to “submit” it, simply *push* your commits via GitKraken. These will now be posted on your account in GitLab and I will be able to follow them from there.

Now on to the methods:

1. We start with addFirst and addLast. In both cases you should simply have to go to the list's sentinel and call that node's addAfter or addBefore method, depending on what you are trying to do. Both functions are one-liners. You will not be able to fully test these methods until after you have also implemented the methods in the next two numbers.
2. Next we will implement the getFirst and getLast methods. These must do the following:

[9] https://en.wikipedia.org/wiki/Double-ended_queue
[10] https://docs.oracle.com/javase/7/docs/api/java/util/Deque.html

- Go to the node that is the successor/predecessor of the sentinel.
- Check if that node equals the sentinel (via ==). This would occur if the list is empty. In this case we must throw a NoSuchElementException according to the documentation of the Deque interface[11].
- Otherwise we return the item that is stored in that node.

3. Next we will implement the removeFirst and removeLast methods. These are very similar to getFirst and getLast except that they use the node's removeSelf method, to remove the node, before they return the item. These methods will all look very similar.

4. Next we will implement peekFirst and peekLast. These are similar to getFirst and getLast, except that instead of throwing an exception they return null in the case where the list is empty.

5. Next we will implement pollFirst and pollLast. These are like removeFirst and removeLast, except that they return null for the case of an empty list instead of throwing an exception.

6. Next we will implement is getNodeAtIndex, which is important for our list iterator to work. You will find it earlier in the file. It is provided an index that starts at 0, and it is supposed to return the actual *node* at that index, or throw an IndexOutOfBoundsException if the index is out of bounds. A skeleton for this method has been provided for you, and you just have to fill in the body of the while loop.

   Note that getNodeAtIndex is a private method. You will not be able to directly check it except for its effect on other methods. In this case, we test it by creating list iterators starting at various indices, as the constructor for DLListIterator uses this method.

7. The next method to implement is size. It is supposed to count how many elements there are in the list. A skeleton for this has also been set up for you, using a **foreach** loop to show you how that works. We are able to have a foreach loop because our class inherits from AbstractSequentialList, which implements the Collection interface[12], which in turn inherits from the Iterable interface[13]. We can do a foreach loop on any object whose class implements the Iterable interface.

8. Lastly, you must implement removeFirstOccurence. It is given an object and searches for that object, and removed the corresponding node if it is found. It returns true if a node was removed. You will want to use the listIterator method to construct an iterator for the list, and follow steps very similar to the already implemented removeLastOccurrence, which uses the reverse iterator (which goes through the list in reverse order).

[11] https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html
[12] https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html
[13] https://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html