

Project 1: Day Planner

DUE: Thursday, September 8th at 11:59pm
Extra Credit Available for Early Submissions!

Basic Procedures

You must:

- Fill out a readme.txt file with your information (goes in your user folder, an example readme.txt file is provided).
- Have a style (indentation, good variable names, etc.) and pass the provided style checker.
- Comment your code well in JavaDoc style and pass the provided JavaDoc checker (no need to overdo it, just do it well).
- Have code that compiles with the command: `javac *.java` in your user directory without errors or warnings.
- For methods that come with a big-O requirement (check the provided template Java files for details), make sure your implementation meet the requirement.
- Have code that runs with the commands: `java PlannerTUI [InputFILE]`

You may:

- Add additional methods and class/instance variables, however they **must be private**.

You may NOT:

- Make your program part of a package.
- Add additional public methods or public class/instance variables, remember that local variables are not the same as class/instance variables!
- Use any built in Java Collections Framework classes anywhere in your program (e.g. no ArrayList, LinkedList, HashSet, etc.).
- Use any arrays anywhere in your program (except the **data** field provided in the **MySortedArray**).
 - When you need to expand or shrink the provided data field, it is fine to declare/use a "replacement" array.
- Alter any method signatures defined in this document of the template code. Note: "throws" is part of the method signature in Java, don't add/remove these.
- Alter provided classes that are complete (**PlannerTUI**).
- Add any additional import statements (or use the "fully qualified name" to get around adding import statements).
- Add any additional libraries/packages which require downloading from the internet.

Setup

- Download the **p1.zip** and unzip it. This will create a folder **section-yourGMUUserName-p1**;
- Rename the folder replacing **section** with the **001, 002, 003, 004, 005** based on the lecture section you are in;
- Rename the folder replacing **yourGMUUserName** with the first part of your GMU email address;
- After renaming, your folder should be named something like: **001-krusselc-p1**.
- Complete the **readme.txt** file (an example file is included: **exampleReadmeFile.txt**).

Submission Instructions

- Make a backup copy of your user folder!
- Remove all test files, jar files, class files, etc.
- You should just submit your java files and your readme.txt
- Zip your user folder (not just the files) and name the zip **section-username-p1.zip** (no other type of archive) following the same rules for **section** and **username** as described above.
 - The submitted file should look something like this:


```
001-krusselc-p1.zip --> 001-krusselc-p1 --> JavaFile1.java
                                     JavaFile2.java
                                     JavaFile3.java
                                     ...
```
- Submit to blackboard.

Grading Rubric

Due to the complexity of this assignment, an accompanying grading rubric pdf has been included with this assignment. Please refer to this document for a complete explanation of the grading.

Overview

There are three major components to this project:

1. Implementing one of the most fundamental data structures in computer science (the dynamic array list).
2. Using this data structure to implement a larger program.
3. Practicing many fundamental skills learned in prior programming courses including generic classes.

We will implement a **sorted** version of the dynamic array list for this project. It still supports the basic features of a dynamic array, which can grow capacity when needed. Furthermore, all items stored inside our sorted dynamic array must be kept in an ascending order. This additional requirement will affect how we could insert new values into the storage and how we are allowed to change the values. We will explain the details below with examples. Make sure you read the whole document before you start coding. The end product will use the sorted dynamic array to implement a day planner. It would allow users to perform various operations in maintaining a single-day planner, including adding an event, moving an event, and changing an event.

Sorted Dynamic Array. A sorted dynamic array is a growable array data structure in which each element is sorted in some order. For this project, we require all elements in the array are sorted in an **ascending** order. The figure below shows an example sorted dynamic array storing integers.

	2	6	6	11	20			
index	0	1	2	3	4	5	6	7

Example 1: A sorted dynamic array with 5 integers, size = 5, capacity = 8

It is required that we always keep the array sorted. Therefore, our implementation of inserting new items and replacing existing items would need to take the ordering requirement into consideration. Here are the detailed rules to follow:

- **void add(T value):** When adding a new item **value** into the array, we need to find an index to insert **value** while keep the array sorted from smallest to largest. A special case is when a repeated value is added. For this project, we require the repeated value be added at the end of the group, i.e. we assign the highest possible index to the new value while keeping the array sorted. Check the example below.

	2	6	6	11	19	20		
index	0	1	2	3	4	5	6	7

After **add(19)**
size = 6, capacity = 8

	2	6	6	6	11	19	20	
index	0	1	2	3	4	5	6	7

After **add(6)**, 6 is a repeated value
size = 7, capacity = 8

Example 2: Array starting from Example 1; array updated after the two consecutive **add()** operations. After each **add()**, latest member added in **red**; shifted members in **blue**.

- **boolean add(int index, T value):** When an intended index is specified to add a value, we need to ensure inserting **value** at **index** will still keep the array sorted. Otherwise, do not update the array and return false. You will also need to check the validity of index. Check examples below.
- **boolean replace(int index, T value):** The replacement operation works similar to **add(index, value)**. Checking must be performed to ensure we have a sorted array after the replacement. Otherwise, do not replace and return false. You will also need to check the validity of index. Check examples below.

	2	6	6	11	20				
index	0	1	2	3	4	5	6	7	

Original array
size = 5, capacity = 8

Operation Applied to <u>Original</u> Array	Operation Effect	Updated Array																
<code>add (4 , 19)</code>	Insert 19 at index 4; return true	<table><tr><td>2</td><td>6</td><td>6</td><td>11</td><td>19</td><td>20</td><td></td><td></td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>	2	6	6	11	19	20			0	1	2	3	4	5	6	7
2	6	6	11	19	20													
0	1	2	3	4	5	6	7											
<code>add (5 , 27)</code>	Insert 27 at index 5 (appending); return true	<table><tr><td>2</td><td>6</td><td>6</td><td>11</td><td>20</td><td>27</td><td></td><td></td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>	2	6	6	11	20	27			0	1	2	3	4	5	6	7
2	6	6	11	20	27													
0	1	2	3	4	5	6	7											
<code>add (2 , 6)</code>	Insert 6 at index 2; return true	<table><tr><td>2</td><td>6</td><td>6</td><td>6</td><td>11</td><td>20</td><td></td><td></td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>	2	6	6	6	11	20			0	1	2	3	4	5	6	7
2	6	6	6	11	20													
0	1	2	3	4	5	6	7											
<code>add (3 , 11)</code>	Insert 11 at index 3; return true	<table><tr><td>2</td><td>6</td><td>6</td><td>11</td><td>11</td><td>20</td><td></td><td></td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>	2	6	6	11	11	20			0	1	2	3	4	5	6	7
2	6	6	11	11	20													
0	1	2	3	4	5	6	7											
<code>add (7 , 30)</code>	Invalid index; throw IndexOutOfBoundsException	Array not updated																
<code>add (4 , 30)</code>	Inserting 30 at index 4 (before value 20) not allowed; return false	Array not updated																
<code>replace (1 , 5)</code>	Replace value at index 1 to be 5; return true	<table><tr><td>2</td><td>5</td><td>6</td><td>11</td><td>20</td><td></td><td></td><td></td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>	2	5	6	11	20				0	1	2	3	4	5	6	7
2	5	6	11	20														
0	1	2	3	4	5	6	7											
<code>replace (2 , 11)</code>	Replace value at index 2 to be 11; return true	<table><tr><td>2</td><td>6</td><td>11</td><td>11</td><td>20</td><td></td><td></td><td></td></tr><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr></table>	2	6	11	11	20				0	1	2	3	4	5	6	7
2	6	11	11	20														
0	1	2	3	4	5	6	7											
<code>replace (5 , 25)</code>	Invalid index; throw IndexOutOfBoundsException	Array not updated																
<code>replace (2 , 5)</code>	Placing 5 at index 2 (after value 6) not allowed; return false	Array not updated																

Example 3: All operations applied to the initial array on top: return and updated array.
New items added in **red**; shifted items in **blue**.

Ordering/comparing. You will define the sorted dynamic array as a generic class storing Comparable items. Remember to use `.compareTo()` (not `>`, `<`, or `==`) to determine the ordering and equivalence of two values. You will need to define **MyTime** and **Event** classes that implement Comparable interface and use them with the sorted dynamic array. We include a brief review of Comparable in the Appendix of this document as a quick reference for you.

Day Planner. The end product of this project is a day planner. With the provided simple textual user interface, users can add events into a 24-hour period (from 0:00 to 23:59). We allow overlap of events but the list of events must be sorted based on their starting times. Users can also delete and update existing events. Check the provided Java template files to see the details of required methods.

In particular, when the starting time of an event is changed, check and make sure:

- If with the updated starting time, the events are still sorted in ascending order of their starting times, do not change the index of the event that we update;
- Otherwise, remove the updated event, then add it back to fix the order.

We include multiple sample runs in a separate PDF file in the project package. Each sample run has a session of planner operations that would help you understand the expected planner behavior better.

Implementation/Classes

This project will be built using a number of classes representing time, event, the generic sorted dynamic array, and the day planner we described in the previous section. Here we provide a description of these classes. Template files are provided for each class in the project package and these contain further comments and additional details. You must follow the instructions included in those files.

- **MySortedArray (MySortedArray.java)**: The implementation of a sorted dynamic array list. You will implement this class as a generic class to practice that concept. It will be used as the storage of events in the day planner.
- **MyTime (MyTime.java)**: The class representing a time with two integer components: an hour within [0,23] and a minute within [0,59]. It implements Comparable interface. Comparison of **MyTime** objects is the basis of ordering events in our day planner.
- **Event (Event.java)**: The implementation of an event with a starting time, an ending time, and a description. It also implements Comparable. The ordering of two events is determined by the ordering of their starting times.
- **Planner (Planner.java)**: The implementation of a day planner. It stores a collection of events in ascending order of their starting times. The planner supports multiple operations for maintenance, including adding a new event, deleting an event, and updating an event.
- **PlannerTUI (PlannerTUI.java)**: A textual user interface class to interact with the user and to help testing your implementation. This class is provided to you and you should NOT change the file.

Requirements

An overview of the requirements is listed below, please see the grading rubric for more details.

- **Implementing the classes** - You will need to implement required classes and fill the provided template files.
- **JavaDocs** - You are required to write JavaDoc comments for all the required classes and methods.
- **Big-O** - Template files provided to you contains instructions on the REQUIRED Big-O runtime for many methods. Your implementation of those methods should NOT have a higher Big-O.

How To Handle a Multi-Week Project

While this project is given to you to work on over about two weeks, you are unlikely to be able to complete this in one weekend. We recommend the following schedule:

- Step 1 (Prepare): First weekend (by 08/28)
 - Complete Project 0 if you haven't.
 - Go over Project 1 with a fine-toothed comb.
 - Read about Dynamic Array List (Ch15 of textbook).
- Step 2 (**MySortedArray**, **MyTime**): Before the second weekend (08/29-09/02)
 - Implement and test methods of **MySortedArray**.
 - Implement and test methods of **MyTime**.
- Step 3 (**Event**, **Planner**): Second weekend(09/03-09/04)
 - Implement and test methods of **Event** and **Planner**.
- Step 4 (**Wrapping-up**): Last week (09/05-09/08)
 - Additional testing, debugging, get additional help.
 - ☺ Also, notice that if you get it done early in the week, you can get extra credit! Check our grading rubric PDF for details.

Testing

The main methods provided in the template files contain useful example code to test your project as you work. You can use command like "**java MySortedArray**" or "**java Planner**" to run the testing defined in **main()**. **Note:** passing all yays != 100% on the project! Those are only examples for you to start testing and they only cover limited cases. Make sure you add many more tests in your development. You could edit **main()** to perform additional testing. JUnit test cases will not be provided for this project, but feel free to create JUnit tests for yourself. A part of your grade *will* be based on automatic grading using test cases made from the specifications provided.

The provided **PlannerTUI** can be run with or without an input file as the command line argument.

- When no file name is provided, the program starts with an empty planner and expects input from keyboard. Check the sample run PDF for details.
- When a valid file name is provided, the program read the file to get a sequence of commands to the planner. It also starts with an empty planner. Each line is one user input / command. Check the sample run PDF for details.
- We provide a number of testing files that you can use with **PlannerTUI** under the folder **input_files**. Check the **readme.txt** in that folder for a brief description for each file.
- Make sure you test more with either keyboard or with your own files.

Appendix: CS211 Background

There are some basic Java concepts that we'll need to understand in order to do this project. It is assumed that you acquired this knowledge in the prerequisite classes (e.g. CS211), but this section outlines some material you may need a refresher on.

Inheritance and Generics

- Java Tutorial Inheritance: <https://docs.oracle.com/javase/tutorial/java/land/subclasses.html>
- Java Tutorial Generics: <https://docs.oracle.com/javase/tutorial/java/generics/index.html>

The majority of the classes in our program will be generic and will be using inheritance. Please make sure these are clear before starting this project.

Comparing Interface

- Java Reference: <https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html>
- Java Reference: <https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html>
- Helpful YouTube Video: <https://www.youtube.com/watch?v=JSvVsOm4oX0>

There are two Java interfaces which help explain to Java how objects can be compared. The **Comparable** interface helps define the “natural” order of custom data types (classes). An example of a “natural” ordering is with numbers. If I gave you a set of numbers and asked you to sort them, you would use their “natural” order to do so.

The **Comparator** interface helps define the order of custom data types that do not have a natural order to them. An example of something that does not have a natural order is a house. If I gave you a set of houses and asked you to sort them, you'd ask for more information (sort by size? price? location? paint color?).

If we make a class and want instances of that class to have the equivalent of **<**, **==**, and **>**, we use one of these two Java features to provide the needed information to the computer. A good video on the different ways to compare instances of a class (including the **Comparable** interface) is linked at the top of this section, but below is a text summary of what we need for this project:

1. Instances can't/shouldn't be compared with the **<**, **>**, or **==** operators.

If we have a class:

```
class Person {  
    String name;  
    int age;  
}
```

And we make two instances:

```
Person p1 = new Person();  
Person p2 = new Person();
```

We can't do the following:

```
if(p1 > p2) {  
    //do something  
}
```

Because Java doesn't know how to “compare” **p1** and **p2** (by **name**? by **age**? by memory location?).

2. If the instances have a “natural” order to them (or only ONE order for this particular project), then we should use the `Comparable` interface.

If people should always be compared by age in this program, then we can have the `Person` class implement the `Comparable` interface to tell Java how to compare people. The `Comparable` interface requires one single method: `compareTo()`.

```
class Person implements Comparable<Person> {
    String name;
    int age;
    public int compareTo(Person p) {
        return this.age - p.age;
    }
}
```

There are two things of note here. First, `Comparable` is generic, and we tell it what the classes can be compared to (e.g. people can be compared with people, so we implemented `Comparable<Person>`). Second, `compareTo()` returns an `int`. The convention is that `p1.compareTo(p2)` returns the following

If logically ...	<code>compareTo()</code> return is...
$p1 > p2$	> 0
$p1 < p2$	< 0
$p1 == p2$	$= 0$

We still can't do this:

```
if(p1 > p2) {
    //do something
}
```

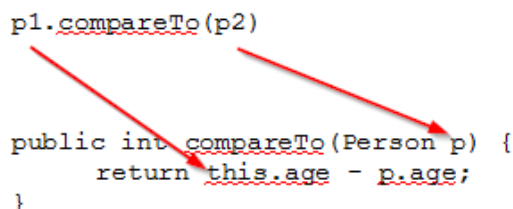
but now we can do this:

```
if(p1.compareTo(p2) > 0) {
    //do something
}
```

In our implementation of `compareTo()` above:

```
public int compareTo(Person p) {
    return this.age - p.age;
}
```

If `p1`'s age was 10 and `p2`'s age was 5, then `p1.compareTo(p2)` would return 5. Note that `p1`'s age becomes `this.age` and `p2`'s age becomes `p.age` due to how we are calling this method:



```
p1.compareTo(p2)

public int compareTo(Person p) {
    return this.age - p.age;
}
```

3. Try this out before continuing with the project.

Make a simple class (like the **Person** class above), have it implement the **Comparable** interface (as shown above), and try calling that method (like above) in a main method. Play around with this to get a good feel for how this works before continuing.

Compiling JavaDocs

- JavaDoc Reference: <http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html>

The amazing Java API documents we've been reading throughout the semester were generated with a tool called **javadoc**. If we have written good JavaDocs, we can "compile" them into a webpage using the **javadoc** command from the terminal. For this project, if the path variable is set correctly for Java, we should be able to compile all the JavaDocs in the user folder by running the following command from the user directory:

```
javadoc -private -d docs *.java
```

If your system doesn't recognize the **javadoc** command, you may need to reconfigure your path (this is part of setting up Java, so you may Google how to do this if you need).

Once we've run the above command successfully, we will have a directory called "docs" in the user folder. In that folder is an index.html file we can open to see a whole website containing our Java documentation for this project! The **javadoc** compiler can also help us debug JavaDocs (it will give warnings when we forget to do certain things).