CSC-223
Lab 1
Work in Pairs

Linear Set and Dictionary

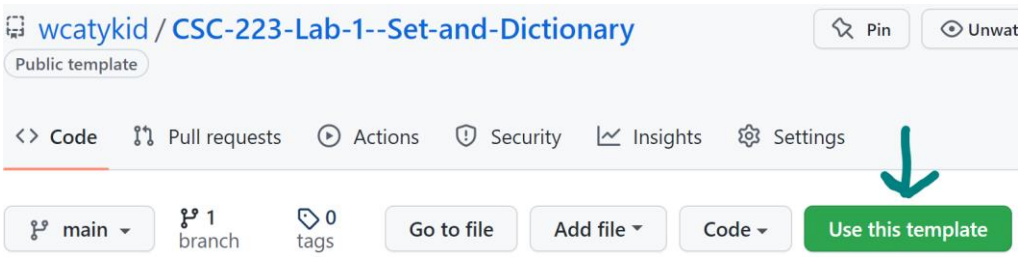
We have two goals for this activity. Our first goal is to have pairs of students working together using a code repository (github) and understand how it facilitates distributed development among members of a team. Our second goal is to implement and test some fundamental, generic data structures in Java: a set and a dictionary.

What You Need to Do: Setup Your Solution

You will notice that this description was linked from the course Moodle page to a file on github. Github is a popular repository tool that acquired its name from the `git` tool. `git` is a version control tool that has been used for decades to facilitate distributed development of software.

Step 1. If you do not have an account on github, create one. We recommend that you use an email address that will stay with you beyond your time at Furman; i.e., a non-Furman email address (although you can always change your primary email address with github). When you interview for jobs, a potential employer may ask you to provide a github account so they can review your portfolio of projects and demonstrate your coding acumen. (We recommend you use github for many purposes, including, but not limited to code. Github has other functionality such as wikis, etc. Some students use it as a repository for all their papers and assignments for their classes.)

Step 2. *Only one person in your group needs to complete this step by creating their own copy of the lab repository.* Go to the lab repository and click on `Use this template`.



You will be directed to a new page in which to create your own repository using my files and directory structure as a starting point. Give the repository a meaningful name like "CSC-223-Lab-1-Linear-Set-and-Dictionary".

Step 3. Download and install Eclipse. Even if you have Eclipse installed, there is a benefit to making sure everyone in your group has the latest version. This will lead to fewer future conflicts.

Step 4. Download the desktop app for github; available for Mac and Windows.

Step 5. Clone your repository in the desktop app; Figure 1 shows how in the app while Figure 2 depicts how to acquire the github repository URL.

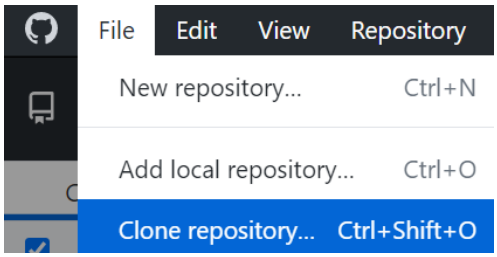


Figure 1: Clone a repository in the github desktop app.

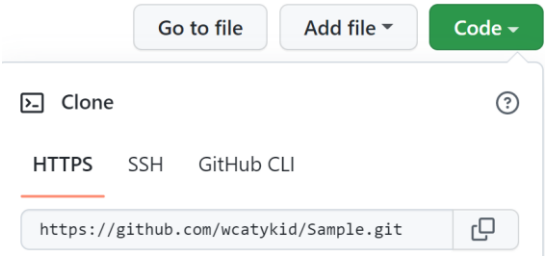


Figure 2: Identifying the URL of your repository as provided on github.

Step 6. Import the project in Eclipse as shown in Figure 3.

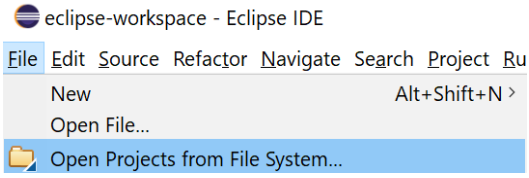


Figure 3: Open the project in Eclipse.

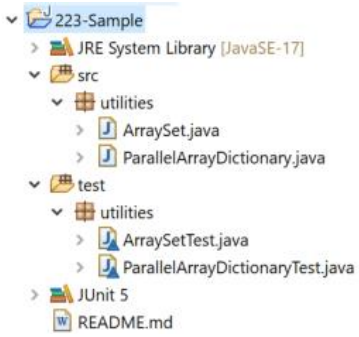


Figure 4: The directory structure and files provided from the github repository you cloned for Lab 1.

In Eclipse, you should have a project that is ready to complete your tasks as shown in Figure 4.

Now, if you make changes in your files in Eclipse, those changes should be reflected in the github Desktop app as shown in Figure 5 through Figure 7.

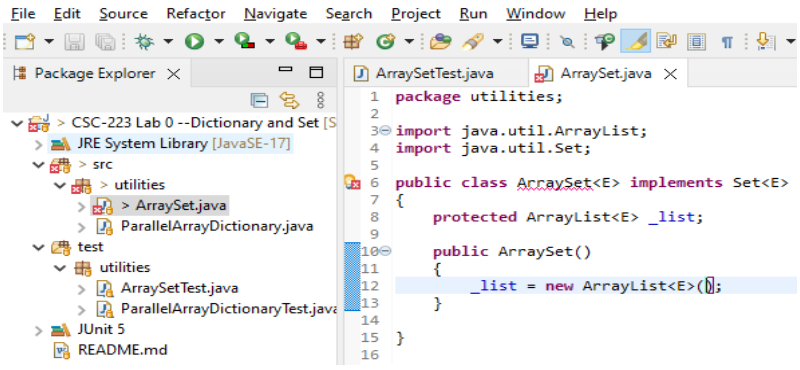


Figure 5: Sample source code additions to the 'blank' starting project.

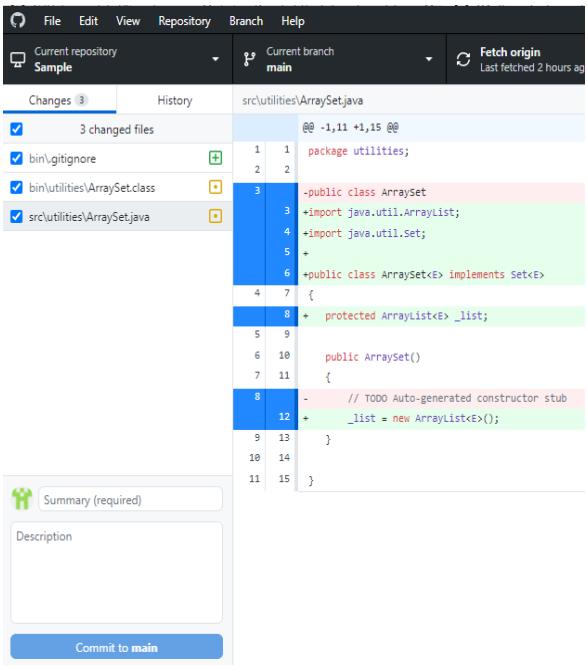


Figure 6: Source code additions reflected in the github Desktop app.

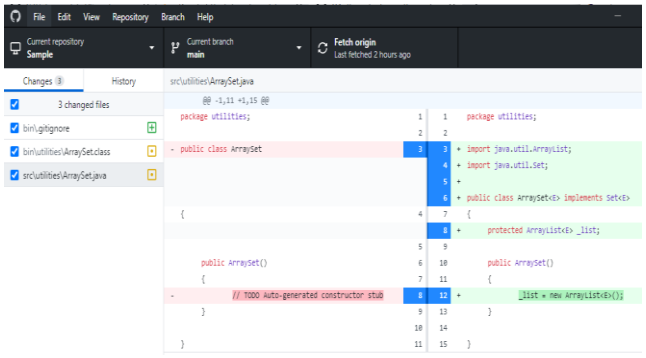


Figure 7: A side-by-side view of source code additions reflected in the github Desktop app.

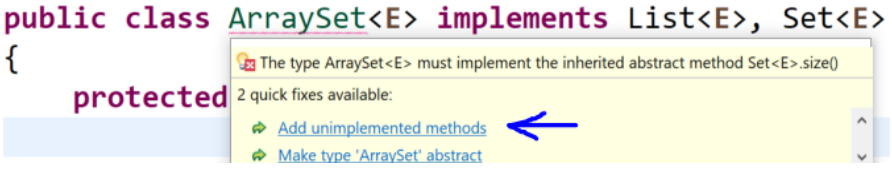
What You Need to Do: The `ArraySet` Class Implementation

In programming, a set is a data structure that guarantees uniqueness of the elements, but says nothing about the order of elements (note: some sets cannot be ordered). You must implement the generic `ArraySet` class according to the following class specification.

```
public class ArraySet<E> implements List<E>, Set<E>
{
    protected ArrayList<E> _list;

    // ... your code goes here
}
```

Although this may not seem very exciting, implementing both the `Set` and `List` interfaces requires that we define many operations. To observe all of the operations you will implement, we can use Eclipse's autocomplete feature where we can *add unimplemented methods*. The result of this operation are method stubs that require your attention to complete implementation. Although the intent of several methods are clear by context, for more information on the interface requirements for each method, please review the Java `Set` and `List` interface API documentation.



All of our `Set` functionality will wrap around an `ArrayList` (object `_list` in the code in Figure 5). This means we can leverage all of the methods `ArrayList` defines. For example, we would implement the `contains` method for `ArraySet` using the contains method for `ArrayList`. This 'wrapping around' implementation is a form of a *design pattern* referred to as a *Decorator* or *Delegator*.

```
@Override
public boolean contains(Object o) { return _list.contains(o); }
```

What You Need to Do: The `ParallelArrayDictionary` Class Implementation

Recall that a dictionary is a data structure that contains `<key, value>` pairs in which a unique `key` is mapped to a (possibly non-unique) `value`.

Your implementation of a dictionary will use parallel containers as shown below. Note that using an `ArraySet` object for the keys is appropriate as it is indexable (because it implements interface `List`) and guarantees uniqueness in keys. And the choice of an `ArrayList` to contain the values is because it is an indexable structure (also implements interface `List`). We say that these are parallel structures because, for example, if `<key, value>` is a valid pair in our dictionary with `key` stored at index `0` in `_keys`, then index `0` in the `_values` list will contain the corresponding `value`.

```
public class ParallelArrayDictionary<Key, Value> implements Map<Key, Value>
{
    // Parallel array-based implementation
    protected ArraySet<Key>    _keys;
    protected ArrayList<Value> _values;

    // ...
}
```

As with `ArraySet`, "Add Unimplemented methods" in `ParallelArrayDictionary` and complete those methods. With our implementations, refer to the `Map` interface documentation provided by the Java API.

What You Need to Do: Testing

You are to implement a set of unit tests for each method of `ArraySet` and `ParallelArrayDictionary` *that does not simply call a delegate method*. For example, we would not need to junit test the `contains` method above since it implements strict delegation by calling the `ArrayList contains` method.

Each class will have its own independent testing file and tests. `ArraySet` will be tested by the `ArraySetTest` class, similarly for our dictionary. Our test classes will always be defined in a *parallel* `test` source folder in Eclipse as shown in Figure 8. It is important to note for future projects that the project has a parallel structure of the packages as well as the classes and classes under test.

The template project provided on github already defines this project structure.

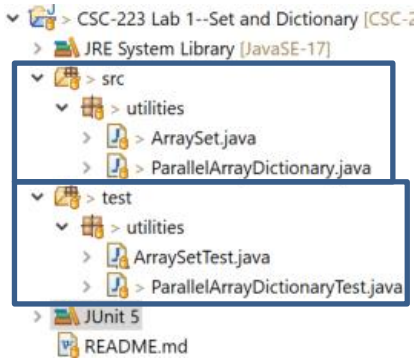


Figure 8: The parallel structure of the project.

You are to implement a set of independent unit tests for `ArraySetTest` and `ParallelArrayDictionaryTest`. You are to test each non-constructor method individually that does not strictly delegate functionality. The methods that require testing are indicated in Figure 9 through the list of junit method stubs; those stubs are provided in the github template.

Figure 9: The junit test methods for our Set and Dictionary classes to complete.

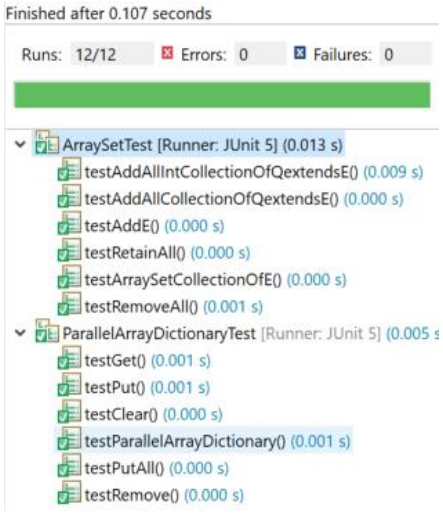


Figure 10: Sample successful junit tests for our Set and Dictionary classes.

When you execute your `junit` tests, ***no output*** should be produced (never print to `System.out` or any other output stream unless it is a file-based logging system); we are seeking only a 'green' output indication in Eclipse. Make sure to use the drop-down menu to show all tests have been executed and are successful as shown in Figure 10; your times should take longer than the ones depicted.

What You Need to Do: Committing Changes

When you make significant changes to your local copy of the project that you want to share with your group of developers, you will need to `commit` your changes to the repository.

Warning: it is never recommended to commit code that (1) does not compile or (2) is known to fail tests.

In the github desktop app, changes will be easily identified in the left pane (Figure 11). Before committing, you must provide a message about the changes you are committing. Your messages should be reasonably descriptive and meaningful retrospectively. This will allow all developers in your group to understand the contents of your commit and, if needed, allow easy information in the future if the commit needs to be `reverted` (undone).

At this point, we are ready to push our changes to the repository. Click the `Push origin` button as shown in Figure 12. Give it a moment and then the changes will be reflected in the repository on github.

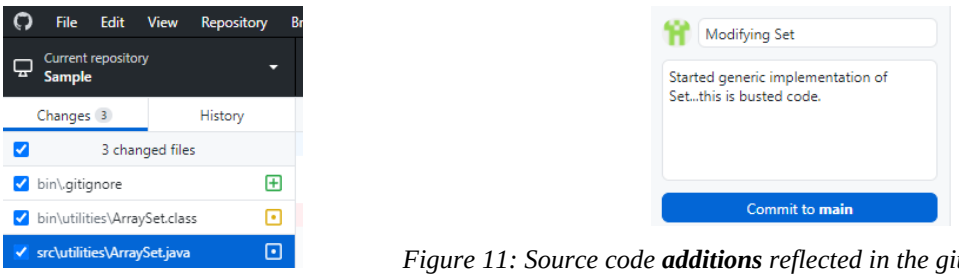


*Figure 11: Source code **additions** reflected in the github Desktop app.*

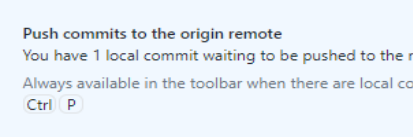


Figure 12: github Desktop application bubble indicating changes are ready to be committed.

What You *May* Need to Do: Resolving Commit Conflicts

If more than one group member modifies a file, *conflicts* may arise. That is, git will need help to identify exactly which segment(s) in a file should be accepted and which should be discarded.

Consider an example where your partner (Developer B) modifies `ArraySet` to include implementations of the constructors as shown in Figure 14, but you have already modified ArraySet to include method stubs for the "unimplemented methods" as shown in Figure 13. If Developer B commits the changes before we do, there will be a conflict. ***Conflicts are not bad. They just need to be resolved.***

```
public class ArraySet<E> implements Set<E>
{
    protected ArrayList<E> _list;

    public ArraySet()
    {
        _list = new ArrayList<E>();
    }

    @Override
    public int size()
    {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public boolean isEmpty()
    {
        // TODO Auto-generated method stub
        return false;
    }
```

Figure 13: A sample of Developer A modifications to `ArraySet`.

```
public class ArraySet<E> implements Set<E>
{
    protected ArrayList<E> _list;

    public ArraySet()
    {
        _list = new ArrayList<E>();
    }

    public ArraySet(Collection<E> collection)
    {
        this();

        for (E item : collection)
        {
            add(item);
        }
    }
}
```

Figure 14: Developer B modifications to `ArraySet`.

We can identify if a conflict has occurred because if we (Developer A) tries to push their changes to the repository, they will receive an error similar to the one shown in Figure 15. *We are required to remove the conflicts* by merging the two versions of the file (one from Developer A and one from Developer B) as indicated in the window in Figure 16.

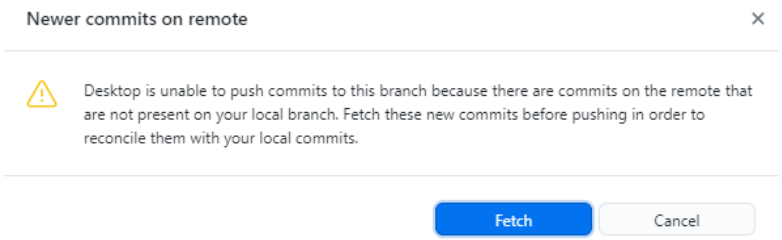


Figure 15: Git saying there is a conflict between your local file(s) and those in the repository.

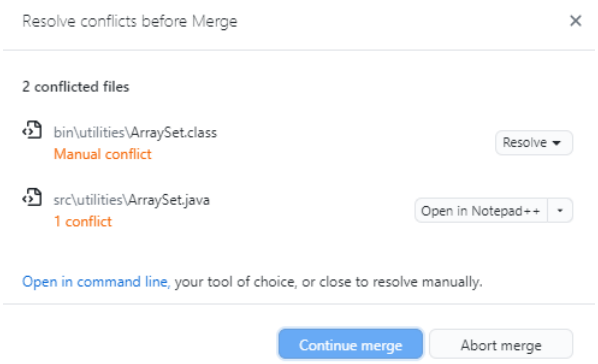


Figure 16: Github conflict resolution screen. We must resolve all file conflicts before merging. We are only interested in merging source code files (.java files); we can ignore others.

As shown in Figure 17, we can use either version of non-text files (e.g., .class, .exe, etc.). For text files such as source code files, we can open each file, in turn, to determine which version we will keep. Although a text editor can be used, it can be more helpful to use other tools such as kdiff3 or DiffMerge. For example, the code in Figure 18 is what you will see when resolving conflicts with a text editor (Notepad++ in this case). 'Resolving conflicts' is a manual process in which you much go through each conflict and decide which version of the code you want to keep. Once all conflicts are resolved, a merge can be completed as shown in Figure 19 with Figure 20 being a screenshot of the repository on github. Success!

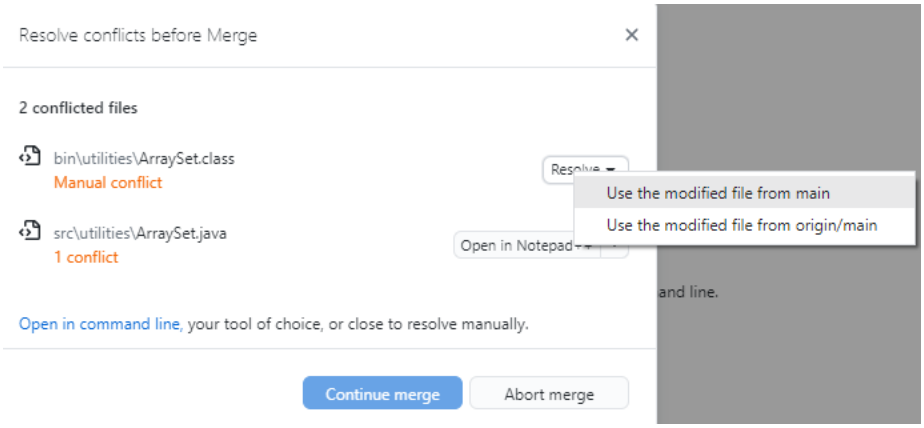


Figure 17: Accepting either version of a non-text file is fine. Source code file conflicts can be opened and resolved in a text editor (or other merge tool).

```
public class ArraySet<E> implements Set<E>
{
    protected ArrayList<E> _list;

    public ArraySet()
    {
        _list = new ArrayList<E>();
    }

<<<<<<< HEAD
    public ArraySet(Collection<E> collection)
    {
        this();

        for (E item : collection)
        {
            add(item);
        }
    }
=======
    @Override
    public int size() {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public boolean isEmpty() {
        // TODO Auto-generated method stub

>>>>>>> efc9c084cac4022be591bc0c1c4a8247788f0d89
}
```

Figure 18: Conflicts identified by git with `ArraySet`. *Conflicts are indicated by blocks (<<<<<<< ... ====). To resolve the conflicts in this file, we can simply delete all of the offending characters (<, =, >) since we want to keep both sets of changes.*

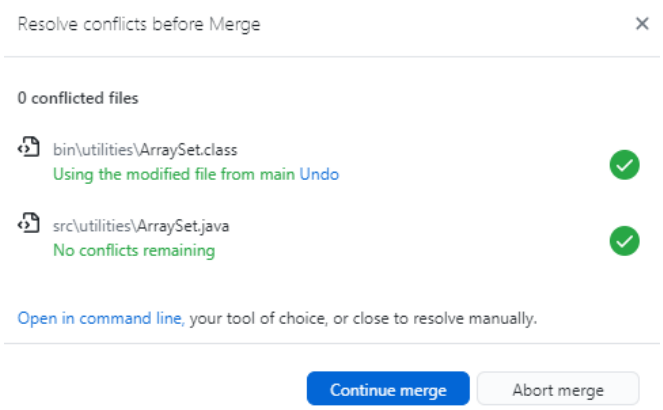


Figure 19: Screenshot of successful resolution of all conflicts in a project.

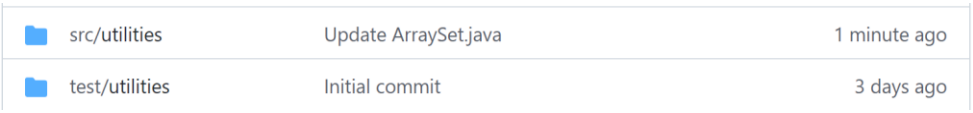


Figure 20: Screenshot of the github once conflicts are resolved and changes are pushed.

Recommendations

We recommend you tackle this project in this order.

1. Setup the project with github. Make sure all group members have access to the repository and can successfully commit changes.
2. Implement `ArraySet` class. Test it method by method by implementing tests in `ArraySetTest`. Communicate with your group; commit as needed.
3. Implement `ParallelArrayDictionary` class. Test it method by implementing tests in `ParallelArrayDictionaryTest`. Communicate with your group; commit as needed.

Commenting

Your code should be well documented, including docstring comments of methods, blocks of code, and header comments in each file. Junit tests should have reasonable and *meaningful* messages printed, if failure occurs (e.g., use the string-based assert functions such as `assertTrue(java.lang.String message, boolean condition)`).

Header Comments

Your program must use the following standard comment at the top of *each source code file*. Copy and paste this comment and modify it accordingly with your files.

```
/**
 * Write a succinct, meaningful description of the class here. You should avoid wordiness
 * and redundancy. If necessary, additional paragraphs should be preceded by <p>,
 * the html tag for a new paragraph.
 *
 * <p>Bugs: (a list of bugs and / or other problems)
 *
 * @author <your name>
 * @date   <date of completion>
 */
```

Inline Comments

Comment your code with a *reasonable amount of comments* throughout the program. Each non-obvious method should have a Javadoc comment that includes information about input, output, overall operation of the function, as well as any limitations that might raise exceptions. Javadoc comments can be inherited.

Each *block* of code (3-4 or more lines in sequence) in a function should be commented. Comments go above and before the code they are referencing.

It is ***prohibited*** to use *long* comments to the right of lines of source code; attempt 80 character-wide text in source code files. Succinct comments (a few words at maximum can be fine).

Submitting: Source Code

For this lab, you will demonstrate your source code to the instructor, in person. Be ready to demonstrate (1) successful execution of all junit tests and (2) the github repository which includes commented source code (see above) and a clear commit history with meaningful commit messages.