Software and Programming II (SP2)
2018/19: Coursework Assignment Three

1 Introduction

- **Submission Deadline: 15 January 2019, 11:55pm GMT**
- Feedback Deadline: 5 February 2019

There are **three** coursework assignments for this module. The coursework assignments contribute to your overall module mark as follows:

- Assignment 1 accounts for 20% to the coursework mark (i.e., 5% of the overall module mark);
- Assignments 2 and 3 account for 40% to the coursework mark each (i.e., 10% of the overall module mark).

Each of the assignments is marked out of 100. The aims of this coursework are:

- To work with recursive data structures.
- To work with object-orientation, inheritance, abstract classes, and interfaces.
- To perform sanity checks on method parameters and to throw exceptions to indicate unsuitable parameter values to the user of your code.
- To give you experience with writing unit tests in JUnit.
- To give you further practice with Git and GitHub.

The code for this coursework is made available to you via the following GitHub Classroom invitation link:

`https://classroom.github.com/a/pImaEPTU`

The assignment is further explained in Section 2 below. Section 3 of this document explains the marking scheme. Section 4 presents the deadlines and submission instructions. Section 5 explains the penalties for late submissions, and Section 6 how the College deals with plagiarism. Section 7 provides additional information on learning resources.

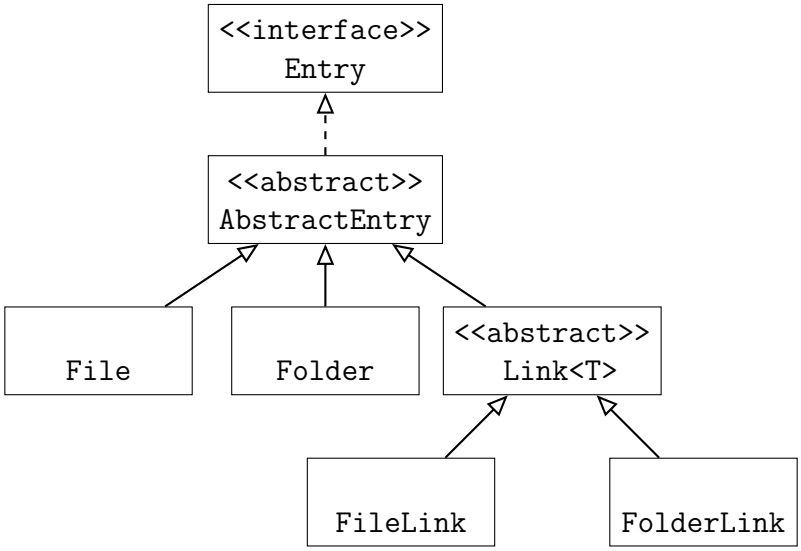


Figure 1: The inheritance hierarchy for our object-oriented data structure in this coursework

2 Description of the work

2.1 File System

In this coursework we are going to develop an extensible object-oriented data structure for managing files, folders, and links to files and folders. Every file will have its name and size recorded. Every folder may contain files, other folders and links. A file (folder) link is just a pointer to another file (respectively, folder). Creating a link has the effect of giving a file or folder multiple names (e.g., different names in different folders) all of which independently connect to the same data on the disk; this causes an alias effect: e.g., if the file is opened by any one of its names, and changes are made to its content, then these changes will also be visible when the file is opened by an alternative name.

Here, part of the code, including an inheritance hierarchy, is already given. You can access the Git repository with the initial files on GitHub via the invitation link in Section 1, similar to Coursework Assignments 1 & 2.

Your task will be to complete the classes so that they implement the desired functionalities using suitable object-oriented concepts and information hiding. Constructors, attributes, and several methods are already present. However, there are still a number of methods pending to be implemented. They are indicated either directly by a "`// TODO`" comment in the code or by the fact that the interface `Entry` requires them, but their implementations are still missing in the inheritance hierarchy.

Figure 1 shows a graphical sketch of the inheritance hierarchy between the classes and interfaces. A box with `<<interface>>` stands for an interface, a box with `<<abstract>>` stands for an abstract class; and a box with neither stands for a concrete class. The dashed and solid arrows stand for the "extends" and "implements" relations, respectively. For example, the concrete class `FileLink` extends the abstract class `Link`, and the abstract class `AbstractEntry` implements the interface `Entry`.

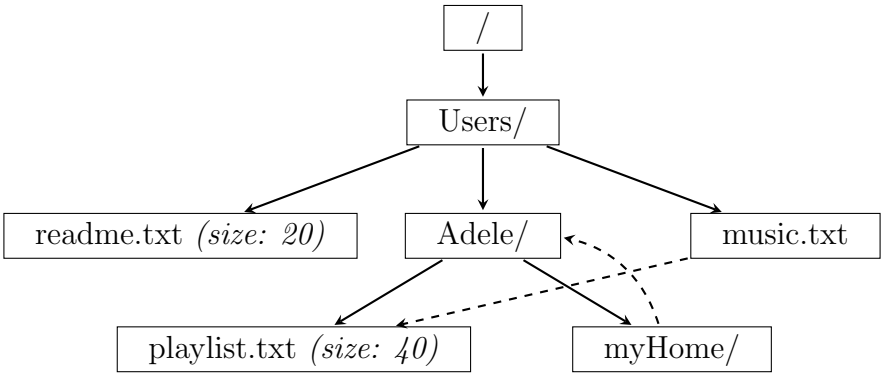
Moreover, we provide a class `Coursework3Main`, which you can use to test your code.

Some further background: On Mac OS X and Linux (both members of the Unix family), there is only one *root folder* for the file system (also called the directory structure): "`/`" In contrast, with the Windows operating system, we may have several such file systems: "`C:`",

"`D:`", "`E:`", ..., where the corresponding root folders are called "`C:\`", "`D:\`", "`E:\`", ...

In this coursework, we shall follow the Mac OS X and Linux convention. In a file system, we can create folders and files inside a folder (this is why there has to be a root folder — we have to start somewhere). We also use links (more precisely, "symbolic links") to files and to folders that can serve as "shortcuts".

For example, we could have a file system with the following structure:



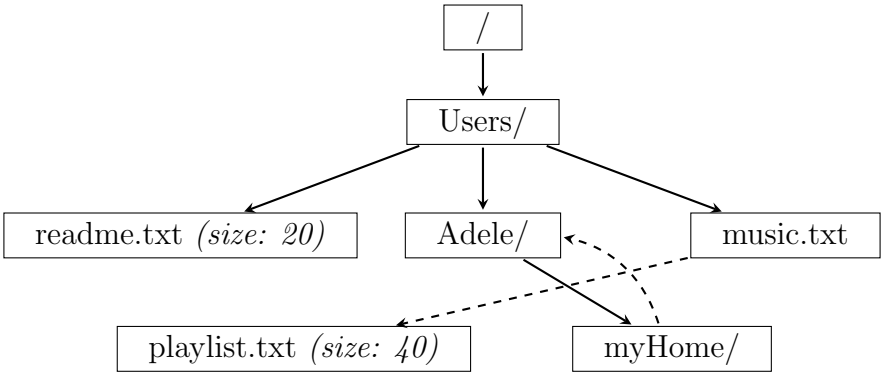
Here, folders (and links to folders) are displayed with a "/" appended to their name, whereas regular files (and links to regular files) are shown without such a trailing "/". A solid edge from a folder to an entry means that the folder contains the entry. For example, the folder "Users/" contains the entry "readme.txt". A dashed edge from a link to a corresponding entry means that the link points to the entry. For example, the file link "music.txt" points to the file "playlist.txt". If a user opens "music.txt" in a text editor and makes changes, the changes will be written to the file "playlist.txt".

The *path* to an entry is obtained by concatenation of all the labels in the (unique!) path via the solid edges from the root folder to the entry. For example, the path to "playlist.txt" is "/Users/Adele/playlist.txt".

We can query an entry for the total *size* of all files at or below the entry. For example, if we query the entry "Users/", we will get 60 (bytes) as the answer: 20 bytes in the file "readme.txt" plus 40 bytes in the file "playlist.txt". And if we query "Adele/" or "playlist.txt", we will get 40 as the answer. If we query "music.txt", we will get 0 as the answer: a link does not consume space (in our model).

We can also ask an entry for its parent folder. For example, the entry "readme.txt" has "Users/" as parent folder. The root folder is special: it has no parent folder. In Java, we represent this via the `null` reference.

Along with adding new entries to a file system and query entries, we can also remove entries (which removes their contents as well, in case of folders). For example, if we remove the file "playlist.txt", we obtain the following result:



The file "playlist.txt" is not outright eliminated, but it is no longer reachable via the solid edges, and we consider it to be "no longer in the file system". The link "music.txt" is thus also considered as invalid after its target "playlist.txt" has been removed.

We can remove entries that are links as well: this removes the link entry from the folder and also its outgoing edge to its "target entry". Note that removing a link does not affect the referenced entry. However, when we remove a folder, this removes also all its contents.

2.2 Given Classes and Interfaces

This subsection provides some background on the classes and interfaces that are given and *should not* be modified (that is, the code that you write in the other classes should work with the *original* classes from this subsection, and markers may in fact completely ignore any modifications that you make in your copies of these classes).

2.2.1 `public interface Entry`

The interface `Entry` describes the methods that each concrete class in the inheritance hierarchy will need to provide along with Javadoc comments.

The interface also contains a string constant `SEPARATOR` used to separate folders and their entries in the text representation of a path to an entry in the file system. It is currently set to `"/"` (your code should not rely on this value, but rather work with `SEPARATOR`).

2.2.2 `public class FileSystem`

The class `FileSystem` contains the root folder of our file system. Thus, we can have several independent file system objects in parallel, and we can also test several independent file systems.

2.2.3 `public class Coursework3Main`

We are providing the file `Coursework3Main.java` in the repository. This class makes use of some of the desired functionalities of our data structure in its `main` method, similar to the description in Section 2.1 You can (and should) test your implementation by running `main`. This provides further clarification for the desired behaviour of the objects. It is a requirement that your implementation compiles with the *unmodified* `Coursework3Main.java` (i.e., all the methods and constructors used by `Coursework3Main.java` should also be present). The file `Coursework3Main.java` contains a comment at the end with the output that its `main` method produces with our implementation.

Note, however, that the "tests" performed by `Coursework3Main` are not meant to be exhaustive — so, even if `Coursework3Main` has the desired outputs, this does not automatically mean that your implementation is necessarily correct for all purposes. Thus, it is a good idea to not only test your code (with further test cases), but also to review it before handing in your solution.

2.3 Classes that You Will Need to Modify

This subsection is about the classes that you will need to modify in this coursework.

We consider concrete classes for the different kinds of entries that a user will actually be able to work with:

- `File`

- `Folder`
- `FileLink`
- `FolderLink`

In addition, we use abstract classes to represent commonalities between (some of) the concrete classes:

- `AbstractEntry`
- `Link<T extends Entry>`

You may need to modify all of these classes.

Recall that the implementation of a method may also be inherited from a superclass where suitable. Then you do not even need to mention that method in the subclass that is supposed to provide the method. Note that your code should not all go into each of the concrete classes, and note that inheriting a method does not mean overriding a method and then calling the superclass method implementation!

2.4 Unit Tests with JUnit

The `main` method of the class `Coursework3Main` uses several of the methods in the classes that implement the `Entry` interface. However, this is not really proper unit testing: unit tests should not rely on a particular execution order, and we cannot check conveniently whether the results of the computations are as expected.

Thus, you need to create a class `EntryTest` as a test suite with the tests listed below, in the format of JUnit version 4 that we used in the lecture. Use the most suitable JUnit assertions for the tested property.

Full marks for the unit testing part can be reached already via the listed tests, but feel free to create more unit tests to check your code. The tests below do *not* achieve test coverage!

1. Test that the root folder of a freshly created file system has size 0.
2. Test that after creating a file "hello" with size 50 in the root folder of a freshly created file system, the *file* actually has size 50.
3. Test that after creating a file "hello" with size 50 in the root folder of a freshly created file system, the *root folder* has size 50.
4. Test that after creating a file "hello" with size 50 in the root folder of a freshly created file system and then removing that file, the root folder has size 0.
5. Test that after creating a file "hello" with size 50 in the root folder of a freshly created file system, that file is in the file system (according to the file's corresponding method).
6. Test that after creating a file "hello" with size 50 in the root folder of a freshly created file system and then removing that file, that file is not in the file system (according to the file's corresponding method).
7. Test that after creating a file "hello" with size 50 in the root folder of a freshly created file system, that file's name is "hello".

8. Test that after creating a file "hello" with size 50 in the root folder of a freshly created file system, the path to that file is "/hello" (using the constant `Entry.SEPARATOR` suitably).

9. Test that after creating a folder with the name "aFolder" in the root folder of a freshly created file system, the path to that folder is "/aFolder/" (using the constant `Entry.SEPARATOR` suitably).

10. Test that after creating a folder with the name "aFolder" in the root folder of a freshly created file system and then creating a folder link with the name "bFolder" to that folder, the target of that link and the original folder are equal.

11. Test that after creating a file "a name" with size 48 as well as a file link "b name" to that file and then removing that file link, the original file is still in the file system.

12. Test that after creating a file "a name" with size 48 as well as a file link "b name" to that file and then removing that file link, the file link is no longer in the file system.

13. Test that creating a folder with the `null` reference as name in the root folder of a freshly created file system throws an `IllegalArgumentException`.

14. Test that creating two files, both called "f", the first of size 20 and the second of size 30, in the root folder of a freshly created file system throws an `IllegalArgumentException`.

2.5 Your Mission

Your mission for this coursework is as follows:

1. *Methods.*

   Think about suitable places in the inheritance hierarchy where you could implement the methods. Then, implement your methods. Note that for methods shared by several classes (in terms of both their headers and their implementations), it is often a good idea to put the implementation into a common superclass.

   For example, the method

   ```
   public FileSystem getFileSystem()
   ```

   of the interface `Entry` is available in all subclasses including `File` and `Folder` thanks to **inheritance** (and can already be implemented in the superclass `AbstractEntry`).

2. *Overriding.*

   You may also need to **override** some methods inherited from superclasses.

3. *Parameter checking.*

   Whenever you take a(n explicit or implicit) parameter in a method where certain parameter values do not make sense (e.g., `null`; see also the method descriptions), detect this at a suitable point in the code and throw an `IllegalArgumentException` to inform the user of your class. For an example, see the public constructor of the class `AbstractEntry`.

2.6 Coding Requirements

- The instance variable declarations present in the classes `AbstractEntry`, `File`, `Folder`, and `Link` are not supposed to be modified.
- Do not add any further instance variables.
- **Do not use the `instanceof` operator in your methods in this coursework.**
- Whenever you override a (concrete or abstract) method from a (direct or indirect) superclass or an interface, always mark this with an `@Override` annotation (see the `toString()` method in `AbstractEntry` for an example).
- Every method (including constructors) should have full Javadoc comments. (However, in this coursework you do not need to write Javadoc comments for methods that you override from a superclass or implement from an interface. Instead, add the `@Override` annotation.) In BlueJ, the "example class" that you get when you create a new class also provides example Javadoc comments. You can use Eclipse's `Source → Generate Element Comment` to get a suitable template with `@param` entries for all method parameters and with `@return` for methods that have a non-void return type.
- **Code style**: One aspect for Java projects is the use of the `this.` prefix before the name of an instance variable or method. **In this coursework assignment**, please use the following style: omit `this.` for calls to instance methods and accesses to instance variables wherever possible (so in your instance methods you would always write `foo()` and `bar` instead of `this.foo()` and `this.bar` unless this leads to the wrong variable being used; see also the code style used in the class `AbstractEntry`). **Note that this code style is *different* from the one required for Coursework Assignment 2!**

  The rationale behind this particular code style is that it is more succinct, and modern IDEs (e.g., Eclipse) usually display instance variables and local variables in different colours to convey the difference.

  (Programmers often spend significantly more time on reading *code, often by other developers, than on* writing *code. This is why many large software projects impose such style guidelines on their contributors so that the code looks uniform and is easy to read for other project members. This requirement is supposed to give you practice for such a setting and to convey that different projects may have different prescribed code styles.)*
- Every class you modify should have your name in the Javadoc comment for the class itself, using the Javadoc tag `@author`. To this end, replace the corresponding `TODO` entries.
- The code template for this coursework is made available to you as a Git repository on GitHub, via an invitation link for GitHub Classroom.

  The idea for the workflow is similar to Coursework Assignments 1 and 2:

  1. First, use a web browser to follow the invitation link for the coursework assignment that is available on the Moodle page of the module. Use your GitHub account to log in.

2. Then, *clone* the Git repository from the GitHub server that GitHub will create for you. Initially, it will contain the following files: `README.md`, `FileSystem.java`, `Coursework3Main.java`, and the files for our inheritance hierarchy.
3. Your task is to enter your name in `README.md` (this makes it easy for us to see whose code we are marking) and to edit the Java source code files according to the requirements of the coursework (i.e., replacing a number of `// TO DO` and dummy implementations of methods with actual code and implementing interface methods at suitable places).
4. **You must also enter the following Academic Declaration into `README.md` for your submission** (see also Section 6):

   > "I have read and understood the sections of plagiarism in the College Policy on assessment offences and confirm that the work is my own, with the work of others clearly acknowledged. I give my permission to submit my report to the plagiarism testing database that the College is using and test it using plagiarism detection software, search engines or meta-searching software."

   This refers to the document at:

   `http://www.bbk.ac.uk/mybirkbeck/services/rules/Assessment%20Offences.pdf`

   A submission without this declaration will get 0 marks.
5. Whenever you have made a change that can "stand on its own", say, "Implemented `getSize()` in `Folder`", this is a good opportunity to *commit* the change to your local repository and also to *push* your changed local repository to the GitHub server.

   As a rule of thumb, in collaborative software development it is common to require that the code base should at least compile after each commit.

- Your source code should be properly formatted. You can use Eclipse's `Source` → `Format` for this purpose. In BlueJ you can use `Edit` → `Auto-layout`.
- *Reminder — use (also auxiliary) methods.* Do not cram everything into one or two methods, but try to divide up the work into sensible parts with reasonable names. Every method should be short enough to see all at once on the screen. (This is probably not a problem in this coursework, also thanks to the way object orientation helps us distribute the code over several classes.)

2.7 Remark

The `toString()`, `equals(Object)`, and `hashCode()` methods from the class `Object` have already been overridden. Do not modify these implementations, and do not add own implementations for these method signatures.

3 Marking Scheme

We aim to award marks according to the following scheme.

1. Code style and comments: **[18 marks in total]**
   (a) Formatting and indentation. **5 marks**
   (b) Consistent use of `myField` and `myMethod()` (without "`this.`" prefix) wherever possible. **5 marks**
   (c) Comments: methods have suitable documentation comments, and `@Override` is used wherever applicable. **8 marks**
2. Methods, with suitable distribution over the available classes: **[54 marks in total]**
   (a) `getName()` **2 marks**
   (b) `getParentFolder()` **2 marks**
   (c) `getSize()` **10 marks**
   (d) `getPath()` **10 marks**
   (e) `isInFileSystem()` **10 marks**
   (f) `remove()` **10 marks**
   (g) `addEntry()` in class `Folder` **10 marks**
3. Unit tests (**2 marks** for each of the **14** required unit test): **[28 marks in total]**

4 Deadlines and Submission Instructions

Submission is through BOTH Moodle AND your GitHub Classroom repository. In Moodle, you need to upload a zip file of the folder containing your working copy and your local Git repository with your modifications to `README.md` and the Java source code files. The GitHub Classroom repository should contain the same commits as your local Git repository.

You should upload the completed assignment on Moodle by

15 January 2019, 11:55pm GMT

(this is Moodle time, not your PC's time; in case you are planning to upload your files whilst at a remote location, make sure you check Moodle's time and take into account the time zone difference). Make sure that the files in your repository on GitHub Classroom correspond to those in your zip file uploaded to Moodle.

It is your responsibility to ensure that the files transferred from your own machines are in the correct format and that any programs execute as intended on Department's systems prior to the submission date.

Each piece of submitted work MUST also have the "Academic Declaration" in `README.md` by the author that certifies that the author has read and understood the sections of plagiarism in College's Policy on Assessment Offences; see `http://www.bbk.ac.uk/mybirkbeck/services/rules/Assessment%20Offences.pdf`. Confirm that the work is your own, with the work of others fully acknowledged. Also include a declaration giving us permission to submit your report to the plagiarism testing database that the College is using.

Reports without the Academic Declaration are not considered as completed assignments and are not marked. The Academic Declaration should read as follows:

> "I have read and understood the sections of plagiarism in the College Policy on assessment offences and confirm that the work is my own, with the work of others clearly acknowledged. I give my permission to submit my report to the plagiarism testing database that the College is using and test it using plagiarism detection software, search engines or meta-searching software."

You should note that all original material is retained by the Department for reference by internal and external examiners when moderating and standardising the overall marks after the end of the module.

We aim to provide you with feedback on your solutions by **5 February 2019**.

5 Late coursework

It is our policy to accept and mark late submissions of coursework. You do not need to negotiate new deadlines, and there is no need to obtain prior consent of the module leader.

We will accept and mark late items of coursework up to and including 14 days after the normal deadline. Therefore the last day the system will accept a late submission for this module is

29 January 2019, 11:55 pm GMT

(this is Moodle time not your PC's time; in case you are planning to upload your files whilst at a remote location, make sure you check the Moodle time and take into account the time zone difference). This is the absolute cut-off deadline for coursework submission.

However, penalty applies on late submissions. Thus, the maximum mark one can get in the coursework is 40 out of 100. If you believe you have a good cause to be excused the penalty for late submission of your coursework, you must make a written request using a mitigating circumstances application form and attach any evidence. Your form should be handed in or emailed to Programme Administrator (with a carbon copy to the module leader and Programme Director) as soon as possible, ideally by the cut-off deadline. This letter/email does not need to be submitted at the same time as the coursework itself but MUST be submitted by

22 January 2019.

Even if the personal circumstances that prevented you from submitting the coursework by the last day are extreme, the Department will not accept coursework after this date. We will, naturally, be very sympathetic, and the Programme Director will be happy to discuss ways in which you can proceed with your studies, but please do not ask us to accept coursework after this date; we will not be able to as there is a College-wide procedure for managing late submissions and extenuating circumstances in student assessment. As soon as you know that you will not be able to meet the deadline, it will be useful for you to inform the module leader. They will be able to advise you on how best to proceed. Another person to speak to, particularly if the problem is serious, is the Programme Director. You will then have the opportunity to discuss various options as to how best to continue your studies.

Further details concerning the rules and regulations with regard to all matters concerning assessment (which naturally includes coursework), you should consult College Regulations at `http://www.bbk.ac.uk/mybirkbeck/services/rules`. Please see the programme booklet for the rules governing Late Submissions and consideration of Mitigating Circumstances and the Policy for Mitigating Circumstances at the College's website `http://www.bbk.ac.uk/mybirkbeck/services/rules`.

6 Plagiarism

The College defines plagiarism as "copying a whole or substantial parts of a paper from a source text (e.g., a web site, journal article, book or encyclopaedia), without proper acknowledgement; paraphrasing of another's piece of work closely, with minor changes but with the essential meaning, form and/or progression of ideas maintained; piecing together sections of the work of others into a new whole; procuring a paper from a company or essay bank (including Internet sites); submitting another student's work, with or without that student's knowledge; submitting a paper written by someone else (e.g. a peer or relative), and passing it off as one's own; representing a piece of joint or group work as one's own".

The College considers plagiarism a serious offence, and as such it warrants disciplinary action. This is particularly important in assessed pieces of work where the plagiarism goes so far as to dishonestly claim credit for ideas that have been taken from someone else.

Each piece of submitted work MUST have an "Academic Declaration" by the student in the file `README.md` which certifies that the student has read and understood the sections of plagiarism in the College Regulation and confirms that the work is their own, with the work of others fully acknowledged. This includes a declaration giving us permission to submit coursework to a plagiarism testing database that the College is subscribed.

If you submit work without acknowledgement or reference of other students (or other people), then this is one of the most serious forms of plagiarism. When you wish to include material that is not the result of your own efforts alone, **you should make a reference to their contribution, just as if that were a published piece of work**. You should put a clear acknowledgement (either in the text itself, or as a footnote) identifying the students that you have worked with, and the contribution that they have made to your submission.

7 Useful resources

Here are some resources on plagiarism, study skills, and time management that can help you to better manage your project and avoid plagiarism.

On Plagiarism

- `https://owl.english.purdue.edu/owl/resource/589/1/`

On Study Skills

- `http://www.bbk.ac.uk/student-services/learning-development`