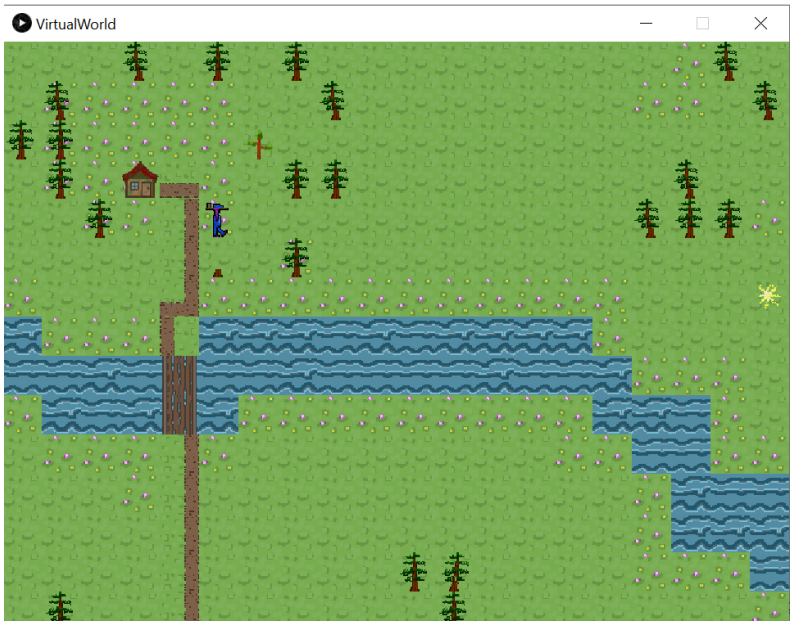
Forest Project (Project 1)



Just as in the real world, characters (people and animals) have both state (i.e., data, e.g., where they are in the world and their mood), and are capable of actions (e.g., walking, eating, etc.). We want our computational objects to encapsulate both data and functions (calledmethods when they are associated with an object).

For this project you will be provided the code for a virtual world program and a UML diagram of the classes used to structure this program.

The program, as given, is written in a non-object-oriented style like that seen in CPE 101. In particular, the vast majority of the functionality is defined as `static` methods in the `Functions` class (with a few methods defined in `EventComparator`, `Point`, and `VirtualWorld`). You should take some time to skim the provided code to get a basic sense of how it is organized.

You might question the quality of this design (which is great!), especially if you have some experience with object-oriented design; you should, however, note that this is a perfectly valid approach (though some parts are intentionally structured for later improvements) that might actually serve you well in a language that does not directly support object-oriented programming, such as C in CPE 357.

Task Overview

This section gives you a high-level overview of what you're supposed to do. Please read this and the following sections **_in full_** before beginning to work on the project.

Download the source code from Canvas.

There are two main parts to the project:

1. The codebase itself

2. A *design document* containing a UML class diagram that describes the structure of the codebase: `ForestProj.drawio`

Your task is to identify the behavior associated with each class (i.e., the behavior exhibited by instances of the class) and move that behavior from the standalone static methods in `Functions.java` to (static or non-static, as appropriate) methods defined within the class. For this assignment, you will not add new functionality (aside from some accessor/mutator methods, only as needed).

You need to perform this refactoring in the design document as well as in the code itself, taking care not to change the functionality. You're encouraged to perform the refactoring in the UML first, and then making those changes to the code. But you are welcome to perform both refactoring simultaneously if you prefer.

Either way, make sure to do the code refactoring _incrementally_. After each change, **test your code** to be sure that nothing is broken. Otherwise, you risk making a mistake, then building on that mistake, such that it is difficult to recover.

The next sections contain more details.

Part 1: Design Task

Your first task is to modify the UML such that the `static` methods in `Functions.java`are moved to appropriate instance methods for the various classes. As you work on this, you may want to refer to the actual code.

Start by locating a copy of the UML diagram of the project as-is. This is available in the source code you downloaded: `ForestProj.drawio`

You are encouraged to develop the UML design document first, however, you are also welcome to simultaneously work on the code refactoring. If you do start refactoring the code, you are encouraged to *implement the refactoring incrementally so that your refactored program executes properly at each step.*

The provided UML diagram was created using the online diagram editor draw.io.

Before doing anything else, make a copy of the file and call it `Methods.drawio`. This `Methods.drawio` file is the one you will be editing as part of this design assignment. To open the file for editing, navigate to draw.io and do `File --> Import from --> Device`. Then choose the `Methods.drawio` file in the file dialog that appears.

To edit the list of attributes and methods for a class, double-click in the class and edit by typing.

Please take a moment to load the given diagram into draw.io and take a look at the general structure of the project. We will be using this code base for the rest of the quarter so please take the time to make friends with it.

Part2: Source Code Refactoring Task

After completion of the first few lab assignment(s) for this course, you should be comfortable with the basics of building and executing Java programs in IntelliJ IDEA.

The provided source code relies on the processing.org API for the graphical interface. To use this library outside of the Processing environment, you will need the library's JAR file for both compilation and execution.

Moving methods

You must refactor the methods from the `Functions` class to move them into the appropriate classes as previously discussed. As each method is moved, you will need to make modifications to the code that uses the method.

Your refactoring should mirror the work done for your design document(UML diagram). If you are refactoring the code at the same time as your UML, make sure any changes you make show up in your UML diagram.

Your refactoring must not add or remove any functionality. Your refactoring may add accessor/mutator methods, but only as needed. The resulting program must work as before.

It is not sufficient to simply move the static methods from `Functions` to the other classes and then continue to invoke them as public static methods. For instance, if you determine that a method works primarily on data within an `Entity` object, then the method must be made non-static and the explicit `Entity` argument will be replaced by the implicit`this`. This modification will necessitate appropriate changes to the invocation of the method.

As an example, moving the following (fake) method into `Entity` will change it as shown.

```
> class Functions
>      {
>  public static void turnAround(Entity entity, int numRotations)
>              {
>                 ... entity.id ...
>              }
>           }

>// invocation of turnAround
>                 turnAround(entity, 20);
```

becomes

```
> class Entity
>       {
>       public void turnAround(int numRotations)
>               {
>                  ... this.id ...
>               }
>       }


>// invocation of turnAround
>           entity.turnAround(20);
```

Tips on Refactoring Methods

You can use the compiler (on the command-line or in the IDE) to help you with your refactoring. In particular, as you make changes, the compiler will flag now invalid uses of moved methods. This serves two purposes. The first and arguably most important, is gaining an understanding of the error messages that the compiler reports and the reasons for such error messages. Nobody enjoys seeing error messages, but quickly interpreting and addressing such errors will improve your workflow.

The second purpose for using the compiler as an aid is that it can quickly identify all parts of a code base affected by a change. This is incredibly beneficial when working with unfamiliar code. (Many IDEs also provide similar support even without explicitly compiling.)

Consider the following more specific tips:

- Examine the data attributes in each class. Draw a graph of dependencies between the classes (in the project) based on the data stored in each (e.g., `WorldModel` relies on `Entity`). Start your refactoring by moving methods into those classes that depend on the fewest other classes.

- In the class you are currently examining, change the access modifier for each data attribute to `private`. Compile the program to determine which methods attempt to access these private attributes.

- If you are using IntelliJ, explore using the `Refactor > Move` feature. Though this feature will not make all the necessary modifications, it will help with the task. Beware that overuse of this feature may interfere with the learning objectives; but using this feature after manually moving a few methods will save you time.

- After moving a method (and removing `static`), remove the target object from the parameter list (and change all uses within the method to `this`, implicitly or explicitly). Compile the program to determine where the method was invoked.

Recall from the design document:

Access Modifiers

In keeping with the principles of encapsulation that we have discussed in class, all data attributes should be `private`, and when possible, `final`. Some constant (`static final`) values. `Point` is another exception to this since each value acts as a constant value akin to an integer.

Methods should also be `private` unless `public` access is necessary (i.e., it is used outside the defining class). For this project, every method should be either `private` or `public` (it is often better to avoid the default of package-protected).

Deliverables

Your project submission will contain the following deliverables:

* The refactored source code

* The `Methods.drawio` UML diagram

* A file called `DESIGN.txt` --- a plain text document justifying your design choices; this should definitely include justifications for methods you did not move from `Functions.java`, but should also justify placement of ambiguous methods (i.e., methods for which it was not obvious to you to which class they belong)

All of this must be submitted through Canvas. Put all the deliverables in a compressed folder and upload it before the due date.