Coding Standards

Purpose: Establish a direct and intuitive agreed upon set of practices that will be consistently followed by all the programmers throughout the java implementation of the product.

Object Oriented Structure

- **New Class Creation:** Class definitions must be created to match the outline in the object oriented design. In java each class is placed in its own file within the source code directory.
- **Inheritance:** When there are multiple features that have common functionality or requirements a new abstract parent class should be created that generically provides this functionality and child classes should inherit from the parent using the extends flag. Classes may also inherit from predefined Java library classes.
- **Private / Public / Protected Flagging:** Methods and attributes within classes can take the property of private, public or protected.
    - **Private:** When an attribute is used only internally in a class or should only be accessed via getter / setter methods it should be labeled as private. Helper methods within a class should also be private.
    - **Public:** A method or attribute that will be used outside the scope of the class should be given the title public.
    - **Protected:** An attribute or method in a parent class that is accessed by a descendant class should be marked as protected.
- **Initialization:** Every class unless it is abstract should have a constructor method where all the attributes are initialized to either default or given values and all the visual components corresponding to that class are created.
- **Main Class:** In Java the main class is the entry point for the program. The gui and all other required components should be initialized in the main class.
- **Multi threading:** Any attribute that may be accessed by multiple threads should be labelled as volatile and only accessed while in a synchronization block.
- **Importing Modules:** All imports should happen at the very beginning of the file and any unused import must be removed to avoid confusion.
- **Error Handling:** A defensive programming tactic should be employed where inputs are verified to be valid before passing them to the logic that will work with them. If an invalid input is given the method must output a warning and return.
- **Method Creation:** A method should be created to perform one specific task that pertains to its name. If 2 tasks are required then 2 methods should be created each for that specific task. Private helper functions to run internal tasks not specified in the design may be created.

Code Commenting

- **Class Documentation:** Below the declaration of each class a block comment will be given that provides a brief overview of the purpose of the class and the function that it performs.
- **Inline Comments:** Inline comments may be created by the programmer if they deem it valuable to provide a note or some further description of what is happening in the code. There is no such thing as self documenting code and inline comments are valuable to others and yourself.
- **Commented Code:** Since every commit is saved to github, redundant or deprecated code should be removed rather than merely commented out.
- **Method Commenting:** Doc strings for each method are not required, but sufficient inline commenting should be provided to allow for quick and easy comprehension of what is happening in any given method.

Naming Conventions

- **All names should succinctly and sufficiently describe the purpose or function of the attribute, class or method**
- **Variables:** snake_case
- **Methods:** camelCase
- **Classes:** PascalCase

File Organization

- All source code should live in the src directory
- All visual aspects of the application should be placed in a components directory
- All backend control classes should be placed in a backend directory
- The maven java build pom.xml file must be placed in the same directory as src and all maven commands should be run from where the project pom.xml exists.

Git / GitHub Standards

Purpose: This is not intended to be a git tutorial, but rather provide a normalized standard for file sharing and version control. Also this document is just a starting point to keep us on the same page, but the actual practices can be negotiated if they are unwieldy or incomplete. It is just helpful if we try to comply by the same version control standards to prevent undesirable surprises. It is fine to change

something about the standard provided that everyone knows about it and consents to it.

Basic git commands

stage git changes: `git add <file1> <file2> ...` or `git add .` for all changes
locally commit staged changes: `git commit -m "message"`
push the changes to the remote GH repo: `git push origin <branch_name>` *pulling changes from the remote GH repo:* `git pull origin <branch_name>` *please note that this will pull and attempt to merge the remote code with yours. Commit your changes locally before doing this.*
fetching code from remote without merging: `git fetch origin <branch_name>`
switch to another branch: `git checkout <branch_name>`
Check up on your local git repo: `git status`
Check for differences between another branch: `git diff <branch_name>`
Look at past commits so that you can revert to them if needed: `git log`
See all other git branches: `git branch -a`

Branches

Purpose: Git branching is super important for allowing many people to work on the same project without breaking things for everyone else. These are the branching standards that we will use throughout this project.
Notes about main: Branches are super useful especially for ongoing or incomplete work. We will use the **main** branch as the production branch where features and attributes of the project will live once completed. The main branch should ALWAYS be stable, anyone should be able to clone the main branch and easily follow simple steps to get it running.
Branching practices: A new branch can be created using the `git branch <branch_name>`. This will also essentially copy all the code and history from your current branch to the new one. If you are going to make a fairly major update to a current aspect of the project or add a new feature to the project it works well to make a new branch named feature_name_wip (wip stands for work in progress). Once the feature has reached a stable mile stone or is complete it can be merged back into the main branch.
Working with existing feature branches: If a feature branch already

exists for something that you are working on, you can make a work in progress branch of that feature and then merge it in once you are ready.

Branch Naming Conventions

Purpose: Using consistent and descriptive naming conventions for branches helps in identifying their purpose and status.
Common Conventions:

1. **Feature Branches:** `feature/<description>` (e.g., `feature/login-page`) *An new component or part of the code. Features should be stable*
2. **Bugfix Branches:** `bugfix/<description>` (e.g., `bugfix/fix-login-error`) *Patch to any fairly major flaw in the code or documents in the project. Minor bugs can just be fixed and commited directly with a descriptive name.*
3. **Documentation Branches:** `docs/<description>` (e.g., `docs/project_description`) *Since much of this project is about software engineering we will be working a lot with documents and planning. The docs branch directory will be used to add to and modify any of our documentation*
4. **Work in Progress (WIP) Branches:** `wip/<description>` (e.g., `wip/new-feature`) *Used to add / work on new components in the project. Works in progress are assumed to be unstable. Once merged back into the desired branch the wip branch can be deleted using* `git branch -d wip/feature_wip` *&&* `git push origin --delete wip/feature_wip`

Merging and conflict resolution

Purpose: This is probably where the most problems are encountered when using git. Using good management of branches along with frequent commits helps in mitigating against nasty merge conflicts that need to be manually resolved.
Merging feature branch into another branch: Once you have a feature that is stable or complete you may need to merge it into another branch.

1. stage and commit all local changes. `git add . ; git commit -m "message"`
2. checkout the branch that you want to merge the feature into. `git checkout <base_branch>`

3. merge the feature branch into the base branch `git merge <branch_name>` *There are plenty of merge strategies that can be applied through git for various purposes, but they can be found pretty easily if needed.*
4. resolve merge conflicts if there are any. *This is usually best done through your ide. If using vscode you can install a git extension and an interface for resolving conflicts should come up if there are any issues.*
5. stage and commit the merge resolutions.

Best practices

1. Commit frequently. After you make a change, commit that.
2. Always commit your changes after working on the project.
3. Please don't commit broken code. Wait until a feature is complete.

Git resources that I like

More notes on best practices for git:
https://gist.github.com/luismts/495d982e8c5b1a0ced4a57cf3d93cf60