

# Umm

A java build tool for novices, also a scriptable autograder, and also provides OpenAI generated feedback.

Repository: [dhruvdh/umm \(github.com\)](https://github.com/dhruvdh/umm)

# 1. Build Tool

- `javac` is not easy to work with.
- Package managers are ideally how you want to work with Java projects.
- IDEs like VSCode expect this.
- Package managers are not novice-friendly.

# 1. Build Tool

- Especially so when you want students to write Unit Tests, and use things like JUnit.
- We also believe the reason labs and assignments do not generally use libraries is that they're difficult to set up -
- Not because there isn't additional educational value in using real-world libraries in labs and assignments.

# 1. Build Tool

```
> umm --help
```

Build tool for novices

Available commands:

run	Run a java file with a main method
check	Check for syntax errors
test	Run JUnit tests
doc-check	Check a file for missing javadoc
grade	Grade your work
clean	Cleans the build folder, library folder, and vscode settings
info	Prints a JSON description of the project as parsed
update	Update the umm command
check-health	Checks the health of the project
reset	Reset the project metadata, and redownload libraries
exit	Exit the program

# 1. Build Tool

It will figure out what files are where, create a source path, class path, and run the requested command.

It will download certain libraries it knows about, and realizes the project needs, like JUnit, as needed.

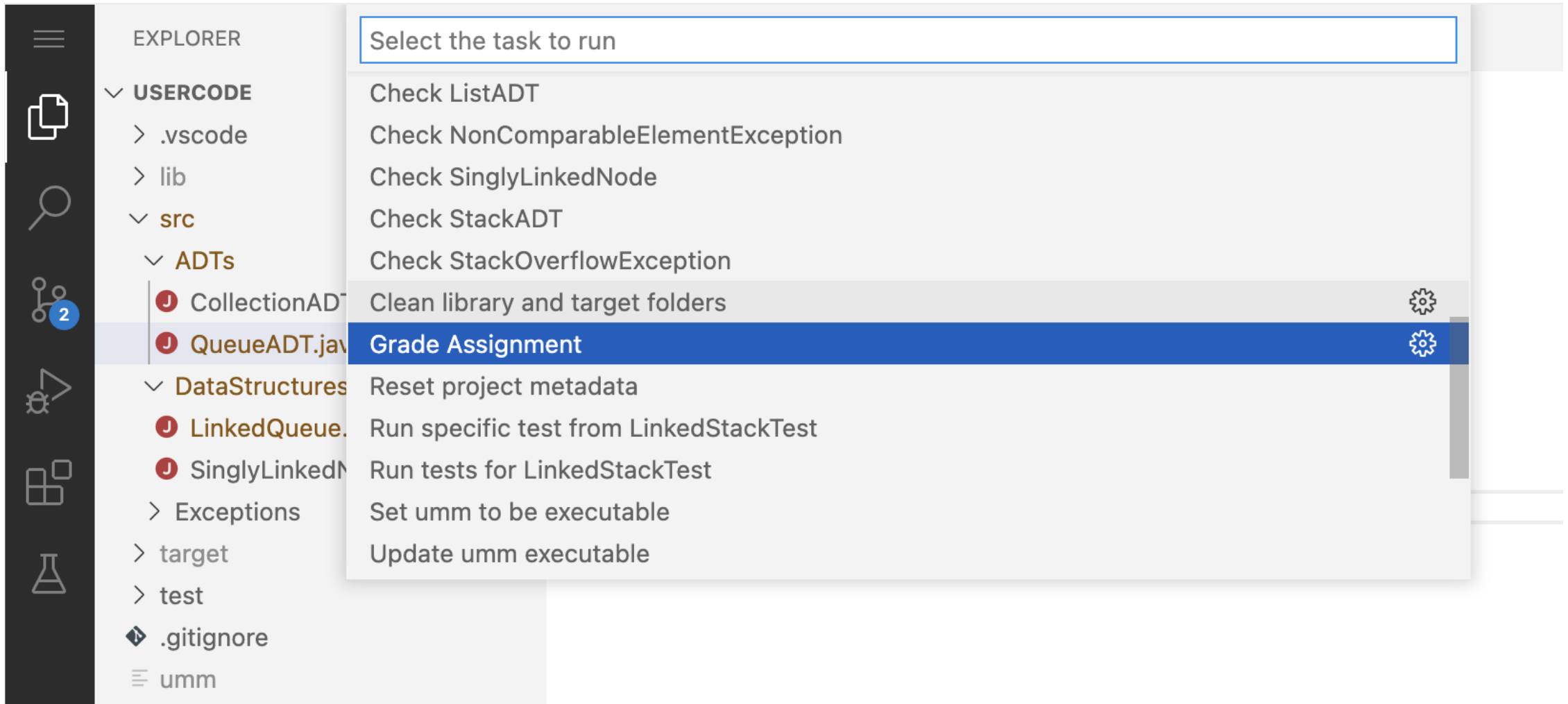
It will create a `.vscode/settings.json` file, and let VSCode all this information. Otherwise, VSCode will just show a million errors.

It will create a `.vscode/tasks.json` file, and expose everything it can do as a VSCode task, so that students don't have to interact with the command line.

# 1. Build Tool

*It is overkill for running a single `Main.java` file, but as recent developments in AI tools have shown, there is very limited value in labs and assignments that are short enough to be worked in a single `Main.java` file.*

# 1. Build Tool



## 2. Auto-grading scripts

A proper, fully-featured embedded scripting language is integrated into the tool.

Objects and functions specific to Java project autograding are exposed to the script.



## 2. Auto-grading scripts

```
let project = new_java_project();

let req_1 = new_docs_grader()
    .project(project)
    .files(["pyramid_scheme.LinkedTree"])
    .out_of(10.0)
    .req_name("1")
    .penalty(3.0)
    .run();

let req_2 = new_by_unit_test_grader()
    .project(project)
    .test_files(["pyramid_scheme.LinkedTreeTest"])
    .out_of(20.0)
    .req_name("2")
    .run();

show_results([req_1, req_2]);
```

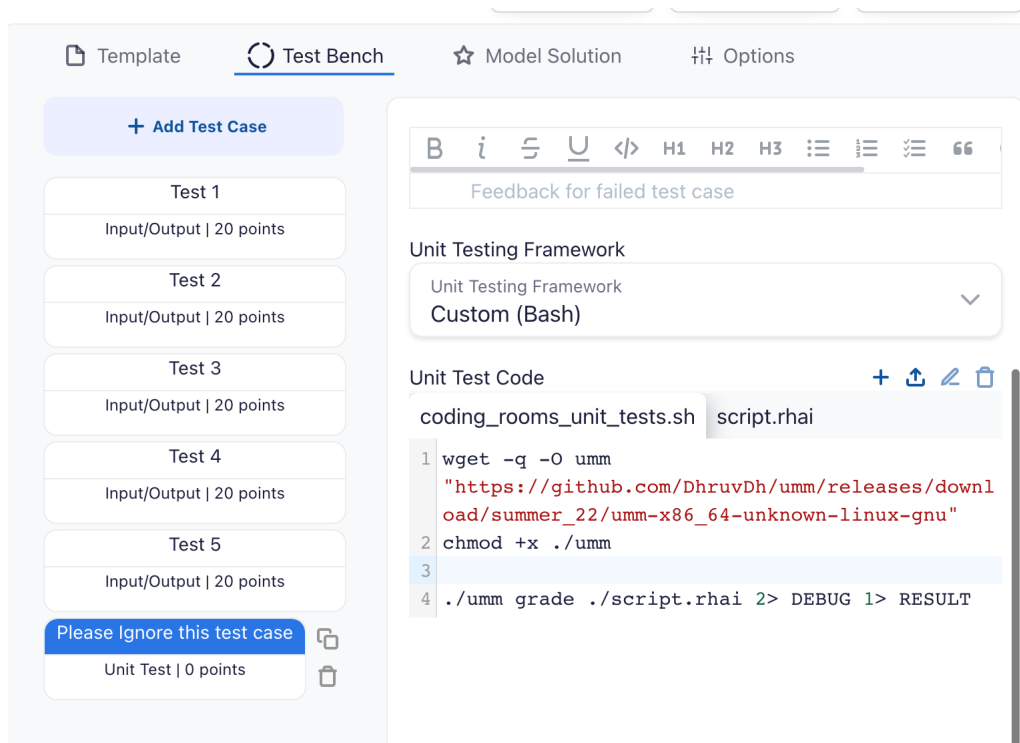
## Output (project had compile errors)

Check javadoc for pyramid_scheme.LinkedTree		
File	Line	Message
./src/pyramid_scheme/LinkedTree.java	15	no description for @param
./src/pyramid_scheme/LinkedTree.java	29	no description for @param
./src/pyramid_scheme/LinkedTree.java	56	Error: unknown tag: T
./src/pyramid_scheme/LinkedTree.java	72	no description for @throws
./src/pyramid_scheme/LinkedTree.java	251	no description for @param
-15 due to 5 nits		

Grading Overview		
Requirement	Grade	Reason
1	0.00/10.00	See above.
2	0.00/20.00	Error running tests.
Total: 0.00/30.00		

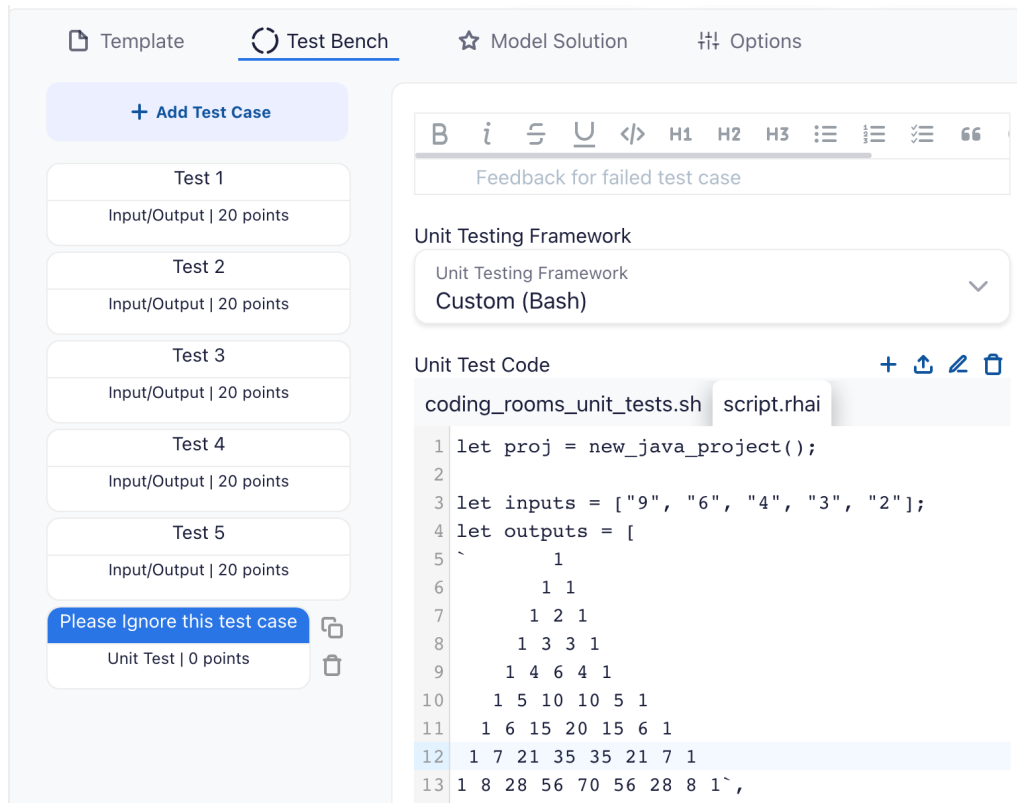
## 2. Auto-grading scripts

Integrates with CodingRooms Custom tests -



## 2. Auto-grading scripts

Individual scripts can be attached to individual test cases, and edited right within the test bench -



The screenshot displays the 'Test Bench' interface. On the left, a sidebar contains a list of test cases: 'Test 1' through 'Test 5', each with 'Input/Output | 20 points', and a 'Unit Test | 0 points' case with a 'Please ignore this test case' button. The main area on the right is titled 'Test Bench' and includes a toolbar with icons for Bold, Italic, Underline, Code, and Headers. Below the toolbar is a 'Feedback for failed test case' section. The 'Unit Testing Framework' dropdown is set to 'Custom (Bash)'. The 'Unit Test Code' section shows a code editor with two tabs: 'coding\_rooms\_unit\_tests.sh' and 'script.rhai'. The code in 'coding\_rooms\_unit\_tests.sh' is as follows:

```
1 let proj = new_java_project();
2
3 let inputs = ["9", "6", "4", "3", "2"];
4 let outputs = [
5     1
6     1 1
7     1 2 1
8     1 3 3 1
9     1 4 6 4 1
10    1 5 10 10 5 1
11    1 6 15 20 15 6 1
12    1 7 21 35 35 21 7 1
13    1 8 28 56 70 56 28 8 1`,
```

## 2. Auto-grading scripts

Available "graders" (constructors are shown) -

1. `new_docs_grader` - checks for missing/invalid JavaDoc
2. `new_by_unit_test_grader` - grades by running JUnit tests
3. `new_unit_test_grader` - grades *student-written unit tests*, by running mutation testing.
4. `new_by_hidden_test_grader` - grades by running hidden tests, downloaded from a URL, and deleted after grading.
5. `new_diff_grader` - Input/output comparison.
6. `new_query_grader` - grades by querying the syntax tree of certain files (ensure student used `substring`, did not import `x`, etc.).

## 2. Auto-grading scripts

Available objects or types that are relevant to grading ([for a language reference see this](#)) -

1. `new_java_project` - a `JavaProject` object, used to represent a Java project.
2. `new_java_file` - a `JavaFile` object, used to represent a Java file.
3. `new_java_parser` - a `Parser` object, that can be used to query the syntax tree of supplied code.
4. `new_grade` - a `Grade` object, used to represent a grade.
5. `new_grade_result` - a `GradeResult` object, used to represent the result of a grading operation. Every grader returns a `GradeResult` when `run()` is called.
6. `new_query` - a `Query` object, that represents tree-sitter queries run on file ASTs.

## 2. Auto-grading scripts

Available functions -

1. `show_results(Array<GradeResult>)` - prints a table of grade results.
2. `generate_feedback(Array<GradeResult>)` - generates a feedback URL for each rubric item, which includes ChatGPT-generated feedback. Students can also interact and ask clarifying questions.
3. `clean()` - cleans the project, removing all downloaded libraries, and deleting the `.vscode` workspace metadata folder, and also the `target` folder containing compiler artefacts.

## 2. Auto-grading scripts - `JavaProject`

A `JavaProject` object represents a Java project.

- `new_java_project()` -> `Project` - creates a new `Project` object. It discovers all Java-related files, also currently downloads certain libraries if detected, updates `VSCode settings.json`, and also creates a `VSCode tasks.json` file.
- `identify(String)` -> `File` - identifies a file in the project, by its name. Returns a `JavaFile` object.
- `files()` -> `Array<String>` - returns an array of all file names in the project.
- `info()` -> prints to stdout a JSON representation of the project, for debugging purposes.



## 2. Auto-grading scripts - `JavaFile`

A file in the discovered project representing any class, interface, or test.

- `new_java_file()` -> `JavaFile` - a constructor, is not meant to be used inside a script. `JavaFile`s should be discovered by the project.
- `check()` - checks for compiler errors, and returns output as a `String`. Also ensures a corresponding `.class` file is present in the target directory after a `check()` completes.
- `doc_check()` -> `String` - asks `javac` for documentation lints using the `-Xdoclint` flag. Returns compiler output as a `String`. There is a parser that can help parse this output which is not currently exposed.
- `run()` - runs the file, and returns stdout/stderr as a `String`.

## 2. Auto-grading scripts - `JavaFile` (cont.)

- `query(String) -> Array<map>` -> accepts a Treesitter query as a string (use backticks for multiline strings), and returns an Array of [Object Maps](#) (dictionary). Each element of the array represents one match, and each object map contains captured variable names as the key, and captured values as the value.
- `test(Array<String>) -> String` - Can be called on JUnit test files. It takes in an Array of strings representing test method names. These test methods must exist within this test file. Passing an empty array will result in all tests discovered being run. Returns output from JUnit as a string.

## 2. Auto-grading scripts - **JavaFile** (cont.)

- `kind() -> JavaFileType` - returns the kind of file (Class, Interface, ClassWithMain, Test)
- `file_name() -> String` - returns the name of the file.
- `path() -> String` - returns the relative path to the file as a string.
- `test_methods() -> Array<String>` - Can be called on JUnit test files. It returns an Array of test method names discovered in the file.

## 2. Auto-grading scripts - Parser

A `Parser` object can be used to query the syntax tree of a Java file. It is not recommended to use this directly, but rather use the `query` method on a `JavaFile` object, or even better, use the `QueryGrader`.

- `new_java_parser()` - a constructor, not meant to be used inside a script. It is ideal if you use `JavaFile`'s `query()`.
- `code() -> String` - returns the source code the parser is working with.
- `set_code(String)` - a setter for the source code the parser is working with.
- `query(String) -> Vec<Dict>` - Currently this method returns a value that cannot be used inside a rhai script, please use `JavaFile`'s `query(String)` instead.

## 2. Auto-grading scripts - `Grade`

A `Grade` object represents a grade.

- `new_grade(float, float) -> Grade` - takes the actual grade received, and the maximum grade as floating point numbers, and returns a `Grade`.
- `from_string(String) -> Grade` - takes a string in this format - `"80/100"` and returns a new `Grade`.
- `grade() -> float` - a getter for the grade received.
- `grade(float)` - a setter for the grade received.
- `out_of() -> float` - a getter for the maximum grade.
- `out_of(float)` a setter for the maximum grade.
- `to_string()` - returns the grade in this format as a string - `"80/100"`.

## 2. Auto-grading scripts - `GradeResult`

A `GradeResult` object represents the result of a grading operation. Every grader returns a `GradeResult` when `run()` is called.

- `new_grade_result()` → `GradeResult` - takes a `Grade` object, and a string representing the feedback, and returns a `GradeResult`.
- `requirement()` → `String` - a getter for the name of the rubric item this `GradeResult` represents.
- `requirement(String)` - a setter for the name of the rubric item this `GradeResult` represents.
- `grade()` → `Grade` - a getter for the underlying `Grade` object.
- `grade(Grade)` - a setter for the underlying `Grade` object.

## 2. Auto-grading scripts - `GradeResult` (cont.)

- `out_of()` -> `float` - a getter for the maximum grade.
- `out_of(float)` - a setter for the maximum grade.
- `reason()` -> `String` - a getter for the "reason" behind the grading.
- `reason(String)` - a setter for the "reason" behind the grading.

## 2. Auto-grading scripts - Query

A `Query` object represents tree-sitter queries that can be run on `JavaFile` s. [Read about queries and captures here.](#)

- `new_query()`  $\rightarrow$  `Query` - a constructor, not meant to be used inside a script. It is ideal if you use `JavaFile` 's `query(String)` .
- `query()`  $\rightarrow$  `String` - a getter for the query string.
- `query(String)` - a setter for the query string.
- `capture()`  $\rightarrow$  `String` - a getter for the capture name.
- `capture(String)` - a setter for the capture name.



## 2. Auto-grading scripts - DocsGrader

A DocsGrader object represents a grader that checks for missing or invalid JavaDoc and penalizes each instance.

- `new_docs_grader()` -> DocsGrader - a constructor, that initializes a new DocsGrader object with default (mostly empty) values.
- `project()` -> Project - a getter for the project this DocsGrader will be run on.
- `project(Project)` - a setter for the project this DocsGrader will be run on.
- `req_name()` -> String - a getter for the name of the rubric item this DocsGrader represents.
- `req_name(String)` - a setter for the name of the rubric item this DocsGrader represents.

## 2. Auto-grading scripts - DocsGrader (cont.)

- `files() -> Array<String>` - a getter for the file names this `DocsGrader` will be run on.
- `out_of() -> float` - a getter for the maximum grade.
- `out_of(float)` - a setter for the maximum grade.
- `penalty() -> float` - a getter for the penalty per missing or invalid JavaDoc.
- `penalty(float)` - a setter for the penalty per missing or invalid JavaDoc.
- `run() -> GradeResult` - runs the grader, and returns a `GradeResult`.

## 2. Auto-grading scripts - DocsGrader (An example)

```
let proj = new_java_project();

let result = new_docs_grader()
    .project(proj)
    .files(["Main.java"])
    .out_of(10.0)
    .req_name("Main.java JavaDoc")
    .penalty(3.0)
    .run();

show_results([result]);
generate_feedback([result]);
```

Repl.it - <https://repl.it.com/@DhruvDhamani/UmmDocsGrader?v=1>

(please fork to play with it)

## 2. Auto-grading scripts - `ByUnitTestGrader`

A `ByUnitTestGrader` object represents a grader that runs JUnit tests on a file, and grades based on the number of tests passed.

- `new_by_unit_test_grader()` → `ByUnitTestGrader` - a constructor, that initializes a new `ByUnitTestGrader` object with default (mostly empty) values.
- `project()` → `Project` - a getter for the project this `ByUnitTestGrader` will be run on.
- `project(Project)` - a setter for the project this `ByUnitTestGrader` will be run on.
- `test_files()` → `Array<String>` - a getter for the names of the tests this `ByUnitTestGrader` will be run on.
- `test_files(Array<String>)` - a setter for the names of the tests to run.
- `expected()` → `Array<String>` - a getter for the names of the expected test methods.

## 2. Auto-grading scripts - `ByUnitTestGrader` (cont.)

- `expected(Array<String>)` - a setter for the names of the expected test methods. For example, if `testAdd` is expected, but is missing in the test file, the grader will return `0`. Unexpected tests found will cause the grader to return `0`. Leaving it empty will unexpected extra tests, and grade all discovered tests.
- `out_of() -> float` - a getter for the maximum grade.
- `out_of(float)` - a setter for the maximum grade.
- `req_name() -> String` - a getter for the name of the rubric item this `ByUnitTestGrader` represents.
- `req_name(String)` - a setter for the name of the rubric item this `ByUnitTestGrader` represents.
- `run() -> GradeResult` - runs the grader, and returns a `GradeResult`.

## 2. Auto-grading scripts - **ByUnitTestGrader** (An example)

```
let proj = new_java_project();

let result = new_by_unit_test_grader()
    .project(proj)
    .test_files(["DataStructures.ArrayStack"])
    .out_of(100.0)
    .req_name("ArrayStack Unit tests")
    .run();

show_results([result]);
generate_feedback([result]);
```

Repl.it - <https://replit.com/@DhruvDhamani/UmmByUnitTestGrader?v=1>

## 2. Auto-grading scripts - `UnitTestGrader`

A `UnitTestGrader` object represents a grader that *grades student-written unit tests*, using [mutation testing](#). Students are penalized for all surviving mutants.

Mutation tests can only be run on a tests suites that pass.

- `new_unit_test_grader()` -> `UnitTestGrader` - a constructor, that initializes a new `UnitTestGrader` object with default (mostly empty) values.
- `req_name()` -> `String` - a getter for the name of the rubric item this `UnitTestGrader` represents.
- `req_name(String)` - a setter for the name of the rubric item this `UnitTestGrader` represents.
- `out_of()` -> `float` - a getter for the maximum grade.
- `out_of(float)` - a setter for the maximum grade.

## 2. Auto-grading scripts - `UnitTestGrader` (cont.)

- `target_test()`  $\rightarrow$  `String` - a getter for the name of the test file *being assessed, written* by the student\*.
- `target_test(String)` - a setter for the name of the test file *being assessed, written* by the student\*.
- `target_class()`  $\rightarrow$  `String` - a getter for the name of the class file *the student is writing unit tests for*.
- `target_class(String)` - a setter for the name of the class file *the student is writing unit tests for*.



## 2. Auto-grading scripts - `UnitTestGrader` (cont.)

- `excluded_methods() -> Array<String>` - a getter for the names of the methods to not mutate.
- `excluded_methods(Array<String>)` - a setter for the names of the methods to not mutate.
- `avoid_calls_to() -> Array<String>` - a getter for the names of classes to avoid mutating.
- `avoid_calls_to(Array<String>)` - a setter for the names of classes to avoid mutating.
- `run() -> GradeResult` - runs the grader, and returns a `GradeResult`.

## 2. Auto-grading scripts - **UnitTestGrader** (An example)

```
let proj = new_java_project();

let result = new_unit_test_grader()
    .req_name("3")
    .out_of(10.0)
    .target_test(["DataStructures.ArrayStackTest"])
    .target_class(["DataStructures.ArrayStack"])
    .run();

show_results([result]);
generate_feedback([result]);
```

Repl.it - <https://replit.com/@DhruvDhamani/UmmUnitTestGrader?v=1>

## 2. Auto-grading scripts - `ByHiddenTestGrader`

A `ByHiddenTestGrader` object represents a grader that runs JUnit tests on a file, and grades based on the number of tests passed.

The Unit test file is downloaded from a remote URL, used to run tests, and then deleted. Students cannot see the tests, and cannot modify them.

- `new_by_hidden_test_grader()` -> `ByHiddenTestGrader` - a constructor, that initializes a new `ByHiddenTestGrader` object with default (mostly empty) values.
- `url()` -> `String` - a getter for the URL of the test file.
- `url(String)` - a setter for the URL of the test file.
- `test_class_name()` -> `String` - a getter for the name of the hidden test class at the URL.
- `test_class_name(String)` - a setter for the name of the hidden test class at the URL.

## 2. Auto-grading scripts - `ByHiddenTestGrader` (cont.)

- `out_of()`  $\rightarrow$  `float` - a getter for the maximum grade.
- `out_of(float)` - a setter for the maximum grade.
- `req_name()`  $\rightarrow$  `String` - a getter for the name of the rubric item this `ByHiddenTestGrader` represents.
- `req_name(String)` - a setter for the name of the rubric item this `ByHiddenTestGrader` represents.
- `run()`  $\rightarrow$  `GradeResult` - runs the grader, and returns a `GradeResult`.

## 2. Auto-grading scripts - **ByHiddenTestGrader** (An example)

```
let project = new_java_project();

let req_4 = new_by_hidden_test_grader()
    .url("https://www.dropbox.com/s/47jd1jru1f1i0cc/ABCTest.java?raw=1")
    .test_class_name("ABCTest")
    .out_of(10.0)
    .req_name("4")
    .run();

show_results([req_4]);
```

Repl.it - <https://replit.com/@DhruvDhamani/UmmByHiddenTestGrader?v=1>

## 2. Auto-grading scripts - `DiffGrader`

A `DiffGrader` object represents a grader that grades based on Input/Output comparisons.

- `new_diff_grader()` -> `DiffGrader` - a constructor, that initializes a new `DiffGrader` object with default (mostly empty) values.
- `project()` -> `Project` - a getter for the project to run the grader on.
- `project(Project)` - a setter for the project to run the grader on.
- `req_name()` -> `String` - a getter for the name of the rubric item this `DiffGrader` represents.
- `req_name(String)` - a setter for the name of the rubric item this `DiffGrader` represents.

## 2. Auto-grading scripts - **DiffGrader** (cont.)

- `out_of()` -> `float` - a getter for the maximum grade.
- `out_of(float)` - a setter for the maximum grade.
- `file()` -> `String` - a getter for the name of the file to run the grader on. Must have a `main` method.
- `file(String)` - a setter for the name of the file to run the grader on. Must have a `main` method.
- `input()` -> `Array<String>` - a getter for the input to pass to the program. Expected input/output pairs must be at corresponding indices.
- `input(Array<String>)` - a setter for the input to pass to the program. Expected input/output pairs must be at corresponding indices.

## 2. Auto-grading scripts - **DiffGrader** (cont.)

- `expected() -> Array<String>` - a getter for the expected output of the program. Expected input/output pairs must be at corresponding indices.
- `expected(Array<String>)` - a setter for the expected output of the program. Expected input/output pairs must be at corresponding indices.
- `ignore_case() -> bool` - a getter for whether the grader should ignore the case when comparing outputs. `true` also ignores whitespace.
- `ignore_case(bool)` - a setter for whether the grader should ignore the case when comparing the output. `true` also ignores whitespace.
- `run() -> GradeResult` - runs the grader, and returns a `GradeResult`.



## 2. Auto-grading scripts - **DiffGrader** (An example)

```
let project = new_java_project();

let input = ["1", "2", "3", "4", "5", "6", "7", "8", "9", "10"];
let expected = ["2", "4", "6", "8", "10", "12", "14", "16", "18", "20"];

let req_5 = new_diff_grader()
    .req_name("Expected output comparison")
    .out_of(10.0)
    .file("Main.java")
    .input(input)
    .expected(expected)
    .ignore_case(true)
    .run();

show_results([req_5]);
```

Repl.it - <https://repl.it.com/@DhruvDhamani/UmmDiffGrader?v=1>

## 2. Auto-grading scripts - QueryGrader

A `QueryGrader` allows you to grade based on the actual program code. Has the student used a particular method or a particular class? Or not used them? Is a particular expression present in the code, etc?

This is done using tree-sitter queries. You can find more information about tree-sitter queries [here](#).

The way the grader is exposed in the scripting language, you shouldn't need to write tree-sitter queries yourself. But they're very easy to write once you get used to them and should be preferred when possible.

The benefit of using tree-sitter queries is that they're widely used, very fast, and let you "ask questions" to a fully formed syntax tree.

## 2. Auto-grading scripts - QueryGrader (cont.)

Example tree-sitter query, that matches local variable declarations that have a type of `int`:

```
(local_variable_declaration
  type: (_) @type
  (#eq? @type "int")
) @body
```

Here, `@type` and `@body` are the names of the captures. The `#eq?` predicate checks if the value of the capture `@type` is equal to the `int`.

In the script, this would be exposed as a `.local_variables_with_type("int")` method. But not everything you might need would be exposed in the script when you need it.

## Auto-grading scripts - `QueryGrader` (cont.)

- `new_query_grader()`  $\rightarrow$  `QueryGrader` - a constructor, that initializes a new `QueryGrader` object with default (mostly empty) values.
- `project()`  $\rightarrow$  `Project` - a getter for the project to run the grader on.
- `project(Project)` - a setter for the project to run the grader on.
- `req_name()`  $\rightarrow$  `String` - a getter for the name of the rubric item this `QueryGrader` represents.
- `req_name(String)` - a setter for the name of the rubric item this `QueryGrader` represents.

## 2. Auto-grading scripts - QueryGrader (cont.)

- `out_of()`  $\rightarrow$  `float` - a getter for the maximum grade.
- `out_of(float)` - a setter for the maximum grade.
- `file()`  $\rightarrow$  `String` - a getter for the name of the file to run the grader on. Must have a `main` method.
- `file(String)` - a setter for the name of the file to run the grader on. Must have a `main` method.
- `queries()`  $\rightarrow$  `Array<Query>` - a getter for the queries to run on the file.
- `reasons()`  $\rightarrow$  `String` - a getter for the reasoning behind this grader. This is shown to the student when they fail.
- `reasons(String)` - a setter for the reasoning behind this grader. This is shown to the student when they fail.

## 2. Auto-grading scripts - `QueryGrader` (cont.)

- `query(String)` - adds a query to the grader. The query is a string that is parsed into a `Query` object. This can be called multiple times to add multiple queries.
- `capture(String)` - adds a capture *to the last query added*, with the given name. This can be called multiple times, once for each corresponding query added. The captures are used to extract information from the query results.
- `filter(Fn(String) -> bool)` - adds a filter *to the last query added*. This can be called multiple times, once for each corresponding query added. The filter is a function that takes a capture value as a string, decides on whether or not to include it, and returns a boolean. If the filter returns `false`, the capture is not added to the list of captures. This can be used to filter out unwanted results, without needing a complex tree-sitter query.

## 2. Auto-grading scripts - **QueryGrader** (cont.)

- `must_match_at_least_once()` - configures the grader such that it should fail if the query doesn't match at least once (length of matches is 0).
- `must_match_exactly_n_times(int)` - configures the grader such that it should fail if the query doesn't match exactly `n` times (length of matches is not `n`).
- `must_not_match()` -> configures the grader such that it should fail if the query matches at least once (length of matches is not 0).

## 2. Auto-grading scripts - **QueryGrader** (cont.)

- `main_method()` - adds a pre-configured query and capture to the grader that captures the main method.
- `class_body_with_name(String)` - adds a pre-configured query and capture to the grader that captures the class body of a class with the given name.
- `method_body_with_name(String)` - adds a pre-configured query and capture to the grader that captures the method body of a method with the given name.
- `method_body_with_return_type(String)` - adds a pre-configured query and capture to the grader that captures the method body of a method with the given return type.



## 2. Auto-grading scripts - **QueryGrader** (cont.)

- `local_variables()` - adds a pre-configured query and capture to the grader that captures the local variable declarations.
- `local_variables_with_name(String)` - adds a pre-configured query and capture to the grader that captures the local variable declarations with the given name.
- `local_variables_with_type(String)` - adds a pre-configured query and capture to the grader that captures the local variable declarations with the given type.
- `if_statements()` - adds a pre-configured query and capture to the grader that captures the if statements.
- `for_loops()` - adds a pre-configured query and capture to the grader that captures the for loops.

## 2. Auto-grading scripts - QueryGrader (cont.)

- `while_loops()` - adds a pre-configured query and capture to the grader that captures the while loops.
- `method_invocations()` - adds a pre-configured query and capture to the grader that captures the method invocations (calls).
- `method_invocations_with_name(String)` - adds a pre-configured query and capture to the grader that captures the method invocations (calls) with the given name.
- `method_invocations_with_arguments(String)` - adds a pre-configured query and capture to the grader that captures the method invocations (calls) with the given arguments.
- `method_invocations_with_object(String)` - adds a pre-configured query and capture to the grader that captures the method invocations (calls) on the given object.

## 2. Auto-grading scripts - `QueryGrader` (cont.)

- `run_query()`  $\rightarrow$  `Array<String>` - All queries are run in sequence, and filtered in sequence. The results are returned as an array of strings, where each string is the value of a capture.
- `run()`  $\rightarrow$  `GradeResult` - Calls `run_query()`, and then checks the results against the configuration of the grader (by default `must_match_at_least_once()` is true). Returns a `GradeResult` object, which contains the results of the grading.

## 2. Auto-grading scripts - QueryGrader (An example)

```
let project = new_java_project();
let req_2 = new_query_grader()
    .project(proj)
    .file("Main")
    .req_name("Query tests")
    .out_of(6.0)
    .reason("Must use int `j` instead of `i`")
    .class_body_with_name("Main")
    .method_body_with_name("F")
    .for_loops()
    .local_variables()
    .filter(|x| {
        print(x); // print the capture to the console, for debugging
        x.contains("j")
    })
    .run();
show_results([req_2]);
```

## 2. Auto-grading scripts - **QueryGrader** (An example, cont.)

It is recommended to run one large tree-sitter query that does everything instead of many small ones. Captures contain parts of code, and ASTs can be misleading when parsing code that is not fully valid.

This can be avoided by changing the underlying algorithm, but I am not sure when I will have time to do that.

It is currently unknown whether this problem will surface in practice, so I am not sure if I want to put in the effort to fix it.

## 2. Auto-grading scripts - `show_results`

Say you decide having `> 70%` total grade is the threshold for passing, then to report results, you can do something like this:

```
let reqs = [req_1, req_2, req_3, req_4, req_5, req_6];
show_results(reqs);

let total = 0.0;
let out_of = 0.0;
for req in reqs {
    total = total + req.grade();
    out_of = out_of + req.out_of();
}

if total > (0.7 * out_of) {
    print("p;" + total.to_int())
} else {
    print("np")
}
```

### 3. Large Language Model Generated Feedback

Every `GradeResult` object has a `prompt` field that we collect during the process of grading.

This `prompt` currently only includes relevant information when a penalty has occurred, or the test has errored out, with all the information on what went wrong.

On calling `generate_feedback(_)`, and passing `GradeResults` that you want to generate feedback for, these prompts are used to print out a URL students can click on to get AI-generated feedback.

# 3. Large Language Model Generated Feedback (cont.)

Example output:

```
generate_feedback([  
    req_1,  
    req_2,  
    req_3,  
    req_4,  
    req_5,  
    req_6  
]);
```

Here's the link from the image -

<https://feedback.dhruvdh.com/e36872bf-9138-49c1-bac7-59516537ae33>

## Test Run Result (4/11/23, 6:37 PM)

0 Passed 1 Failed 0 Needs Review

0 / 100 Points

Final Grade (0 / 100 Points)

Failed ^

Feedback:

### Understanding Your Autograder Results

- For explanation and feedback on `JavaDoc for ArrayStack` (refer rubric), please see this link - <https://feedback.dhruvdh.com/e08cd704-edf6-4ce8-b23f-ba24bc1a6311>
- For explanation and feedback on `ArrayStack Hidden Tests` (refer rubric), please see this link - <https://feedback.dhruvdh.com/e36872bf-9138-49c1-bac7-59516537ae33>

Summary:

[Show Unit Test Code](#)

Grading Overview		
Requirement	Grade	Reason
JavaDoc for ArrayStack	8.00/20.00	See above.
ArrayStack Hidden Tests	0.00/80.00	Error running tests.
Total: 8.00/100.00		



# 3. Large Language Model Generated Feedback (cont.)

The way feedback is collected is different for each grader.

Currently, the most mature "context gathering" for `prompt` construction is for the `ByUnitTestGrader` and `ByHiddenTestGrader`.

Here, we collect information on failed tests, find parts of the project referenced in the stack trace, and then share that as a prompt.

You can see an example here -

## Instructor Message

You cannot see all of the student's submission as you are an AI language model, with limited context length. Here are some snippets of code the stacktrace indicates might be relevant: :

• Lines 24 to 30 from `ArrayStackHiddenTest` -

```
@Test
public void testSize() {
    assertEquals(0, stack.size());
    try {
        stack.push(1);
        assertEquals(1, stack.size());
        stack.push(2);
```

Method body for ``DataStructures.ArrayStack#size``:

```
@Override
public int size() {
    return top + 1;
}
```

Method body for ``DataStructures.ArrayStack#push``:

```
@Override
public void push(T element) throws StackOverflowException {
    if (size() == buffer.length) {
        expandCapacity();
    }
    buffer[top] = element;
}
```

### 3. Large Language Model Generated Feedback (cont.)

Students can also interact with their feedback, currently only in situations where the feedback isn't perceived as helpful.

Clicking on one of these buttons will create a new message to ChatGPT, explaining in more helpful prompts what the student is looking for. The student can also add additional details before clicking on these buttons.

#### Not satisfied with response?

You can request new feedback by clicking on one of the following buttons that best describes your situation. If you want to share additional notes, you can also type them in the text box below.

Your suggestions are too broad and vague.

I don't understand.

Your suggestions work, but I don't understand why.

The changes you suggested are unnecessary or already implemented in my submission.

Request alternate explanation/solution.

This explanation seems incorrect.

Optionally provide additional notes here before clicking a button above.  
Please keep it short, otherwise it might error out.

### 3. Large Language Model Generated Feedback (cont.)

Here, I added a message `What is ++top?` to the earlier prompt

(<https://feedback.dhruvdh.com/e36872bf-9138-49c1-bac7-59516537ae33>), and click on the `I don't understand` button.

Here is the full feedback that was generated - <https://feedback.dhruvdh.com/e052af2d-9eb2-4f82-99a6-8322ef2f1fa9>

The student can keep chatting, or share this URL with the instructional team and they can chime in.

#### ArrayStack Hidden Tests

0.00/80.00

Error running tests.

#### AI Feedback

Done!

`++top` is a shorthand notation for incrementing the value of the variable `top` by 1. It is equivalent to writing `top = top + 1`.

For example, if `top` has the value `3`, then `++top` will increment `top` to `4`.

This notation is commonly used in programming to increment or decrement the value of a variable by 1.

#### Not satisfied with response?

You can request new feedback by clicking on one of the following buttons that best describes your situation. If you want to share additional notes, you can also type them in the text box below.

Your suggestions are too broad and vague.

## Limitations (in no particular order)

- Clicking `Check Answer` on Coding rooms regularly takes 20-30 seconds to run, potentially due to unnecessary downloads or `umm` and libraries.
- Running the `Grade assignment` task in CodingRooms, is much faster, and usually about 3-5 seconds to complete.
- Seems to take about 20-40 minutes or so for a TA to create and verify grading scripts for individual assignments.

## Future work

- Add Python support, a way to make breaking changes in a non-breaky way. Keep improving context gathering for better feedback.
- A "code review/reflection questions" style feedback for successful submissions that is a result of multiple prompts chained - might take ~5 mins or more to finish, per submission.