Scribble End User Guide

31 October 2022, v1.0

1 Introduction

Scribble is a command line tool for testing and auto-marking of Processing sketches. Processing is a simplified Java based language for learning and prototyping, created by Ben Fry, Casey Reas and volunteers. This project is intended to assist educators with grading Processing based assignments.

Section 3 of this document details usage instructions for students and tutors who will be running tests, but will be provided with the test file. Section 4 describes the format of the test file for test writers and assignment writers.

2 Prerequisites

Scribble is written in the Java processing language, and requires the Java Development Kit (JDK) version 17 or greater to be installed. Note tha the Java Runtime Environment (JRE) is not sufficient to run Scribble, as it requires a working Java compiler to be installed on the system. An open source distribution of the JDK is available for free at adoptium.net.

Scribble also depends on the processing-java tool and the Processing core library, which is included with the download of Processing, available for free at processing.org.

3 Command Line Interface

Scribble is provided as a single jar file. This file should be placed on the local drive, and the directory should be opened in a terminal or command prompt. Scribble can be executed by running the following command:

```
java -jar Scribble.jar
```

The following command line arguments can be appended to the command:

`--silent` No messages are printed to the terminal. This argument is mutually exclusive with `--quiet` and `verbose`.

`--quiet` Only warning, error or fatal errors are printed to the terminal. This argument is mutually exclusive with `--silent` and `verbose`.

`--verbose` All message types, including debug messages, are printed to the terminal. This argument is mutually exclusive with `--silent` and `--quiet`.

`--sketch` *path* Scribble runs in single sketch mode. This argument must be followed by a path to a valid sketch. A Processing sketch is a directory containing a Processing source (.pde) file. This argument is mutually exclusive with `--sketchbook`, but one of either `--sketch` or `--sketchbook` is required.

`--sketchbook` *path* Scribble runs in multi sketch mode. This argument must be followed by a path to a directory containing valid sketches. A Processing sketch is a directory containing a Processing source (.pde) file. That is, this argument should be followed by a path to a directory of directories of Porcessing source files. This argument is mutually exclusive with `--sketch` , but one of either `--sketch` or `--sketchbook` is required.

`--testfile` *path* The path of the test file to be executed. A test file is a Java source file (.java) that conforms to the template given in section 4 below. This argument is required.

`--outfile` *path* Specifies the path the the output file.

`--format` *path* Specifies the format of the output file. This argument is option, but if it is specified, `--outfile` is required. Valid values for this argument are *CSV*.

4 Test File Format

A test specification file is a plaintext Java source file (.java). The contents of the file must be valid Java, but you do not need to compile the file ahead of time, as Scribble will perform this step for you. The file must contain a single class of the same name as the file itself, and this class must extend `scribble.api.TestPlan`. The class must implement the method `run()` from `scribble.api.TestPlan`. Inside the run method, the following methods are available to control Scribble and run tests:

`Scribble()` Returns an instance of the Scribble API for you to use. Most methods used to define tests belong to the Scribble instance.

`Scribble.forAll(Consumer<TestCandidate> c)` The passed consumer is executed against every sketch loaded by Scribble. In the case where the application is running in single sketch mode, the consumer is executed only once. In the case where the application is running in multi sketch mode, the consumer is executed for every sketch sequentially.

It is recommended, for readability, to pass a lambda function to this method.

`TestCandidate.runFor(int n)` Defines the number of frames the sketch will be executed for. Once the frame counter reaches the given integer, the sketch is destroyed, and any remaining tests scheduled for later frames will never be executed.

`TestCandidate.onAfterSetup(int n, BooleanSupplier f)` Executes the BooleanSupplier f immediately after the sketch's `setup()` method was run. The return value of the BooleanSupplier is recorded in the results for the submission.

`TestCandidate.onBeforeDraw(int n, BooleanSupplier f)` Executes the BooleanSupplier f once the sketch has reached frame number n, but before sketch's `draw()` method is called. The return value of the BooleanSupplier is recorded in the results for the submission.

`TestCandidate.onAfterDraw(int n, BooleanSupplier f)` Executes the BooleanSupplier f once the sketch has reached frame number n, immediately after the sketch's `draw` method is called. The return value of the BooleanSupplier is recorded in the results for the submission.

`TestCandidate.onBeforeExit(int n, BooleanSupplier f)` Executes the BooleanSupplier f immediately before the sketch is destroyed. The return value of the BooleanSupplier is recorded in the results for the submission.

`TestCandidate.findField(String s)` Returns a `TestField` object representing the matching field. Use this method to compare the values of variables in the sketch to expected values.

`message(String s)` Prints the string s to the standard output, if the application is not running in silent mode.

See the /example directory in the project's root for an example of a test specification file.