Project 3: A Unit Testing Framework

COMP 121, Spring 2021

Main project due: Mon, Apr 5 @ 11:59pm

Introduction

In this project, you will develop a number of useful testing framework components. First, you will implement your own version of JUnit. Second, you will extend your JUnit to support QuickCheck-style random test generation (though to make this gradable we won't use randomness).

You will undoubtedly need to use java.lang.reflect in this project. Note that **there are no shared test cases for this project**.

Here is a code skeleton for this project.

Part 1: Implementing JUnit

Your first task is to implement the following method in class `Unit`:

```
   public static HashMap<String, Throwable> testClass(String name);
```

Given a class name, this method should run all the test cases in that class. The return value is a map where the keys of the map are the test case names, and the values are either the exception or error thrown by a test case (indicating that test case failed) or `null` for test cases that passed.

Here are the rules your test execution engine should follow:

- Test cases are those methods annotated with `@Test`. We've defined this annotation for you already.
- Test cases are executed in alphabetical order (as defined by `String`'s `compareTo` method).
- If there are any methods annotated as `@BeforeClass`, they should be executed once before any tests in the class are run. If there are multiple `@BeforeClass` methods, they are executed in alphabetical order.
- If there are any methods annotated as `@Before`, they should be run before each execution of a test method. Multiple `@Before` methods should be run in alphabetical order.
- Methods annotated `@AfterClass` and `@After` are analogous to `@BeforeClass` and `@Before`, except they are run after the test methods.
- The `@BeforeClass` and `@AfterClass` annotated methods should be run even if there are no `@Test` methods in the class. (But not `@Before` or `@After` methods.)
- `@BeforeClass` and `@AfterClass` can only appear on static methods. If they appear on an instance method, `testClass` should throw an exception (any exception).
- A method can have only one annotation among `{@Test, @BeforeClass, @Before, @AfterClass, @After}`. If a method has more than one such annotation, `testClass` should throw an exception (any exception).
- `testClass` should catch all throwables from invoking test methods and return them in its result. However, it should not catch any throwables raised in methods annotated as `@BeforeClass`, `@Before`, `@AfterClass`, or `@After`. (Such exceptions can be caught and then rethrown, wrapped in a runtime exception.)
- You can assume that any methods annotated with `@Test`, `@Before`, and `@After` are public instance methods that take no arguments. You can assume that `@BeforeClass` and `@AfterClass` only appear on public methods, and you should check that they only appear on static methods.

After implementing the core JUnit functionality, your next step is to implement a fluent assertion API, in which conjunctions of assertions about objects are written using method chaining. For example, using this interface, we should be able to write

```
  String s = ...;
  Assertion.assertThat(s).isNotNull().startsWith("COMP");
```

meaning that `s` is not null and begins with `"COMP"`.

We've started you off by creating a class `Assertion` defining several `assertThat` methods, but you'll need to create other classes to represent the result of calling the different `assertThat` methods. Here's how your interface should work:

- `assertThat(Object o)` should return an object that you can invoke the following methods on.
    - `isNotNull()` - raise an exception (any exception) if `o` is `null`, otherwise return an object such that more of the methods in this chain can be called. The rest of the methods have the same return pattern, so below we only state the conditions in which an exception is raised.
    - `isNull()` - raise exception if `o` is not `null`.
    - `isEqualTo(Object o2)` - raise exception if `o` is not `.equals` to `o2`.
    - `isNotEqualTo(Object o2)` - raise exception if `o` is `.equals` to `o2`.
    - `isInstanceOf(Class c)` - raise exception if `o` is not an instance of class `c`.
- `assertThat(String s)`
    - `isNotNull()`, `isNull()`, `isEqualTo(o)`, and `isNotEqualTo(o)` as with `assertThat(Object o)`. The `isEqualTo(o)` and `isNotEqualTo(o)` methods should take `Object` as an argument.
    - `startsWith(String s2)` raises an exception if `s` does not start with `s2`.
    - `isEmpty()` raises an exception if `s` is not the empty string
    - `contains(String s2)` raises an exception if `s` does not contain `s2`.
- `assertThat(boolean b)`
    - `isEqualTo(boolean b2)` raises an exception if `b != b2`.
    - `isTrue()` raises an exception if `b` is false.
    - `isFalse()` raises an exception if `b` is true.
- `assertThat(int i)`
    - `isEqualTo(int i2)` raises an exception if `i != i2`.
    - `isLessThan(int i2)` raises an exception if `i >= i2`.
    - `isGreaterThan(int i2)` raises an exception if `i <= i2`.

Part 2: QuickCheck for JUnit

QuickCheck is an automated program testing technique developed originally for Haskell. The idea of QuickCheck is that the programmer specifies a test case (called a *property*) that takes some parameters. The QuickCheck infrastructure runs that test case repeatedly with random choices of parameters. For example, using the implementation for this project, we will be able to write the following property:

```
@Property
public boolean absNonNeg(@IntRange(min=-10, max=10) Integer i) {
  return Math.abs(i.intValue()) >= 0;
}
```

If such a property is defined inside a class whose name is passed to the `Unit` method

```
  public static HashMap<String, Object[]> quickCheckClass(String name);
```

then your code will call `absNonNeg` with many different input integers ranging from -10 to 10, inclusive. For the first one for which `absNonNeg` returns `false`, `quickCheckClass` will add a mapping from the method name (`"absNonNeg"`) to the array of arguments for which the method returned `false` or threw a `Throwable`. Otherwise, if the property runs until termination with only `true` return values, the `"absNonNeg"` will be mapped to `null`. Then `quickCheckClass` will run the next property in the class.

Here is the basic setup for `quickCheckClass`:

- It should run all the `@Property` methods in class `name`, in alphabetical order. Note that this is completely separate from test cases. You can assume, without checking, that no method is annotated as both `@Test` and `@Property`.
- A property is deemed to have failed if it returns `false` or throws any `Throwable`.
- Unlike `testClass`, the `quickCheckClass` method returns a map that has the arguments that lead to the failure, but does not indicate what the failure was (i.e., it doesn't include a `Throwable` in the map).
- The first failing/throwing arguments to the property are stored in the map. After the first list of failing/throwing arguments is found, the property is not executed again.
- All properties are run **at most 100 times**. They may run fewer times depending on limits of the search, as described below.
- All property argument types must be annotated in some way, as follows.
    - `Integer` arguments must be annotated with `@IntRange(min=i1, max=i2)`, indicating the minimum integer value and the maximum integer value (both inclusive) for the argument. You can assume max is greater than or equal to min.
    - `String` arguments must be annotated with `@StringSet(strings={"s1", "s2", ...})`, indicating the set of strings for the argument. You can assume the set of strings in non-empty, and that there are no duplicates in the list of strings.
    - `List` arguments must be annotated with `@ListLength(min=i1, max=i2)`, indicating the minimum and maximum (inclusive) list lengths for the argument. The type `T` must itself be annotated appropriately for its range. For example, `@ListLength(min=0, max=2) List<@IntRange(min=5, max=7) Integer>` indicates an argument with lists of length 0 to 2 containing integers from 5 to 7, e.g., `[], [5], [6], [7], [5,5], [5,6], [5,7], [6,5], [6,6], [6,7], [7,5], [7,6], [7,7]`.You can assume max is greater than or equal to min, and that the number of possible elements for `T` is greater than or equal to the number of elements specified by `max`.

- `Object` arguments must be annotated with `@ForAll(name="method", times=i)`, where `method` is the name of the (public, no argument, instance) method in the property's class that will be called to generate `i` values for the argument. (You can assume the method exists and that the times count is positive.) For example:

```
@Property public boolean testFoo(@ForAll(name="genIntSet", times=10) Object o) {
  Set s = (Set) o;
  s.add("foo");
  return s.contains("foo");
}

int count = 0;
public Object genIntSet() {
  Set s = new HashSet();
  for (int i=0; i<count; i++) { s.add(i); }
  count++;
  return s;
}
```

- No other argument types for properties are allowed. (Note that in the previous bullet point, `Object` really means an argument explicitly notated as `Object`, and not anything else.)

When running a property, your `quickCheckClass` method should call it will **all possible combinations of arguments** up to a maximum of 100 runs. The order in which you run the tests does not matter. (We will almost always be testing your code with properties for which there are fewer than 100 possible argument combinations.)

What to turn in

Put all your code in the code skeleton we have supplied, and upload your solution using Gradescope.