# Unit Testing and Test-Driven Development

Today, we start with unit testing, the practice of using small input/expected output pairs to make sure each part of our code works as intended. We then look at a broader software development process called test-driven development (TDD) and see how GenAI tools can be leveraged in TDD.

## Content Learning Targets

*After completing this activity, you should be able to say:*

- I can explain the benefits of using testing frameworks such as JUnit over manually calling methods and printing results.
- I can construct appropriate, high-quality unit tests for small software systems.
- I can describe the test-driven development (TDD) framework, compare it with alternatives, and discuss its strengths and weaknesses.
- I can integrate GenAI code generation tools with TDD and evaluate the resulting implementations with respect to object-oriented design principles.

## Process Skill Goals

*During the activity, you should make progress toward:*

- Identifying good, thorough test cases for writing unit tests. (Problem Solving)

### Facilitation Notes

TODO: facilitator notes **First hour, before quiz:** Model 1 introduces and motivates JUnit tests using a simple rectangle example class. Model 2 provides guidelines for creating unit tests and walks through the JUnit creation process for the BadFrac example.

**Second hour:** Model 3 introduces the test-driven development (TDD) framework, and Model 4 explores genAI-assisted TDD.

Key questions: #6, #11, #18, #25

Source files: *rectangle/BasicRectangle.java*, *rectangle/BasicRectangleTest.java*, *badFrac/BadFrac.java*

# Model 1 Unit Testing

In previous assignments, we have provided you with unit tests (using the JUnit Java framework) to help you verify that your code is working as expected. Today, we look at how (and why) to create this unit tests from scratch.
Take a look at the files ***BasicRectangle.java*** and ***BasicRectangleTest.java*** in `src/rectangle` and answer the following questions.

## Questions (15 min) **Start time:**

**1**. Examine the methods in ***BasicRectangleTest.java***. What details are present that you do not see in typical source code methods, such as those in ***BasicRectangle.java***? What do you ***not*** see here that you expect to find in source code classes?

The methods have an annotation above them, `@Test`. They also call `assertEquals()` methods which are imported from `org.junit.Assert`. There are no explicit fields or constructors.

**2**. Look up a dictionary definition of "assert". How does it apply to the `assertEquals()` methods you see in JUnit tests? Why do you think the JUnit creators chose "assert" over other words?

The "assert" definition from Merriam-Webster is "to state or declare positively and often forcefully or aggressively". Since the test case fails immediately when its first assert fails, throwing an `AssertionError`, it makes sense to use a word with such a forceful connotation (instead of "check", "test", or similar).

**3**. Run ***BasicRectangleTest.java*** in your IDE as JUnit tests. Which tests fail, and which tests pass?

`testCalculatePerimeter()` and `testToString()` pass; the others, `testCalculateArea()` and `testCalculateDiagonalLength()`, fail.

**4**. Search the JUnit 4 API for the definitions of the `assertEquals` methods. (Hint: Look at the `import` statement at the top of ***BasicRectangleTest.java***.) Which versions of `assertEquals` are we using here to test `BasicRectangle`?

We are using `assertEquals(long expected, long actual)` to compare integer values, `assertEquals(Object expected, Object actual)` to compare String values, and `assertEquals(double expected, double actual, double delta)` to compare double values.

**5**. In the `assertEquals` calls within `testCalculateDiagonalLength()`, what does the third parameter do? Experiment with setting it to different values. Can you adjust this parameter in a way that makes `testCalculateDiagonalLength()` fail earlier?

This "delta" parameter sets the tolerance for the floating-point comparison. Changing the first delta from `0.0001` to `0.00001` causes the assertion to fail.

**6**. Instead of having all of this extra syntax and overhead for JUnit tests, we could just print expected and actual values to the console and compare them. Why do you think using unit test frameworks is a best practice?

IDEs can format JUnit results in an easy-to-read, easy-to-navigate manner. Printouts would require analyzing all lines and spotting subtle differences, such as a single-character difference between long strings.

# Model 2 Creating Unit Tests

When creating unit tests, we want to be thorough and consistent. Just like we have object-oriented design principles to guide the design of our source code, we have some guidelines to follow for unit tests.
Good unit tests are:

- small, testing one thing at a time.
- thorough, covering common cases as well as edge/boundary/special cases (zero, `null`, etc.).
- safeguards against (re)introducing bugs. (When you find and fix a bug, add a unit test that would have revealed that defect, just in case later changes bring it back.)

Poor unit tests might:

- provide illegal input(s) (e.g., a negative value when the method ***requires*** a positive input).
- test multiple things at once (e.g., several different methods).

## Questions (20 min) Start time:

**7**. Look back at the unit tests in ***BasicRectangleTest.java***. Applying the above criteria, assess the quality of these test cases. What are they doing well? How could they be improved?

Pros: testing one method at a time, covering a few different cases, providing only valid inputs.
Cons: tests different rectangles within each method, maybe better to separate; does not test edge cases like length/width of zero.

**8**. In the `src/badFrac` folder, briefly skim ***BadFrac.java*** to discern the purpose of the class and its available methods.

**9**. Following the instructions or demo for your IDE, create JUnit tests for the `isReduced()` and `add(BadFrac incoming)` methods. Rename these test methods to `testIsReduced()` and `testAdd()`. (Link to IntelliJ instructions)

**10**. Split into pairs (or solo, if your team only has three today). Add several `BadFrac` objects as fields in the `BadFracTest` class. Use a variety of parameters so that you can cover common and edge cases in your unit tests. Then, add calls to appropriate `assert` methods inside one of `testIsReduced()` or `testAdd()`, making sure both are covered by someone on your team. (Hint for `testIsReduced()`: In addition to `assertEquals`, JUnit provides `assertTrue` and `assertFalse`.)

**11**. (**Regroup.**) Review the test cases produced by your teammates for the other method. Applying the guidelines at the start of this model, are these test cases on target? Why/why not?

**12**. Run your test cases as a team and inspect the results. If you wrote good test cases, then they should ***not*** all pass. As a team, use the results to fix the bugs in ***BadFrac.java***.

# Model 3 Test-Driven Development (TDD)

While working on a fun side project, we often want to jump into writing code right away (spending as little time on the design phase as possible) and maybe later think about testing our code. For more serious work, such as when a hospital or aircraft needs our code to function correctly all the time, we should consider the technique of ***Test-Driven Development*** (TDD). The TDD process is:

a) Write a new unit test (that fails) for a not-yet-implemented system requirement.

b) Write just enough code to pass the new test.

c) Refactor to better follow object-oriented design principles.

d) Repeat for other system requirements.

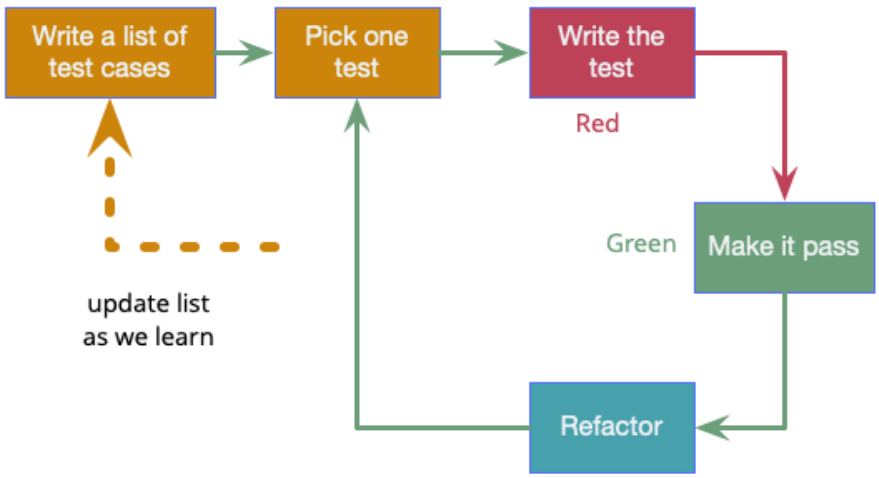


Figure 1: A flowchart of Test-Driven Development by Martin Fowler.

Continuing with our `BadFrac` class, we will practice TDD.

## Questions (20 min) Start time:

**13**. Suppose we are asked to add a new feature to our `BadFrac` class: multiplication. In particular, our new method header should be `public BadFrac multiply(BadFrac incoming)`. Add this method to `BadFrac.java`, with only one line:

```
throw new UnsupportedOperationException("Not yet implemented");
```

**14**. Using the same IDE test generation process as in Model 2, create a `testMultiply()` method in ***BadFracTest.java***.

**15**. Add some `assert` statements in `testMultiply()`, following our guidelines from Model 2.

**16**. Run your new `testMultiply()` test case. Do you get the results you expect?

The first `assert` statement should fail because of the `UnsupportedOperationException`.

**17**. Splitting into pairs (or solo if three), implement the `multiply` method, adjusting it until it passes your unit tests. Then, compare your solution with your teammates' solution(s).

**18**. What do you think are some advantages and disadvantages of TDD? Explain.

See Wikipedia for a nice list. Some pros: clarifies requirements, increases confidence and coverage, many find it boosts productivity. Some cons: more code volume (lots of lines of tests), more maintenance required, learning curve, may incentivize overcomplicated code.

# Model 4 TDD with genAI

One of the challenges with TDD is the time overhead incurred by writing test cases one at a time, then context switching to implementation and refactoring. Since genAI tools have shown promise with code generation tasks and tend to hallucinate less often when provided with a ground truth to work from, some have suggested that genAI could speed up TDD. For example, this workshop paper experiments with collaborative and fully-automated TDD using genAI tools. In this activity, we'll try out genAI-assisted TDD and discuss what to watch out for when using genAI tools in this way.

## Questions (20 min) Start time:

**19**. Add a method stub for `subtract` in ***BadFrac.java***, giving it an appropriate signature and JavaDoc comment. Add the same line as in Model 3:

```
throw new UnsupportedOperationException("Not yet implemented");
```

Expected signature: `public BadFrac subtract(BadFrac incoming)`

**20**. Based on the stated requirements of the `BadFrac` class, especially its constructor, what requirement(s) should we state for the `subtract` method?

The `BadFrac` class represents nonnegative fractions, so we should require that `incoming` is less than or equal to `this`.

**21**. Brainstorm test cases as a team, and add a JUnit test in ***BadFracTest.java*** with your test cases. Run your JUnit test to make sure it fails in the expected way.

**22**. ***Split into pairs*** (or stay as a group of three).

- In one pair (or on one machine), implement the new feature manually.
- In the other pair (or on another machine), provide a genAI tool of your choice with ***BadFrac.java*** and ***BadFracTest.java***, then prompt it to implement the `subtract` method.

TODO: add link to example genAI conversation

**23**. (**Regroup.**) Compare the manual and genAI implementations. Do both pass your JUnit tests? Which implementation seems better, and why?

Answers will vary.

**24**. If time allows, repeat the TDD process for a `divide` method, swapping manual/genAI pairs.

**25**. We might try further integrating genAI into TDD by having the genAI tool write test cases, too. Do you think this is a good idea? Discuss.

One perspective: Having genAI write test cases might be dangerous, because it might not interpret the system requirements correctly. And then if you run the genAI-generated code and see it passes the genAI-generated tests, you might not review the tests and code as thoroughly as you should.

**26**. Skim the "Conclusions" section of the workshop paper mentioned at the top of Model 4. What warnings do the authors include about using genAI with TDD?

The genAI tool might change tests to match buggy code, rather than the other way around. The developer should carefully supervise the quality of the code produced.