Joshua Crotts

Learning Java

A Test-Driven Approach

May 15, 2024

Springer Nature

To my Viola.

Preface

A course in Java programming is multifaceted. That is, it covers several core concepts, including basic datatypes, strings, simple-to-intermediate data structures, object-oriented programming, and beyond. What is commonly omitted from these courses is the notion of proper unit testing methodologies. Real-world companies often employ testing frameworks to bullet-proof their codebases, and students who leave university without expose to such frameworks are behind their colleagues. Our textbook aims to rectify this delinquency by emphasizing testing from day one; we write tests before designing the implementation of our methods, a fundamental feature of test-driven development.

In our book, we *design* methods rather than write them, an idea stemming from Felleisen's *How to Design Programs*, wherein we determine the definition of our data, the method signature (i.e., parameter and return types), appropriate examples and test cases, and only then follow with the method implementation. Diving straight into a method implementation often results in endless hours of debugging that may have been saved by only a few minutes of preparation. Extending this idea into subsequent computer science courses is no doubt excellent.

At Indiana University, students take either a course in Python or the Beginner/Intermediate Student Languages (based on Scheme/Racket), the latter of which involves constant testing and remediation. Previous offerings of the successor course taught in Java lead students astray into a "plug and chug" mindset of re-running a program until it works. Our goal is to stop this once and for all (perhaps not truly "once and for all," but rather we aim to make it a less frequent habit) by teaching Java correctly and efficiently.

Object-oriented programming (and, more noteworthy, a second-semester computer science course) is tough for many students to grasp, but even more so if they lack the necessary prerequisite knowledge. Syntax is nothing short of a different way of spelling the same concept; we reinforce topics that students should already have exposure to: methods, variables, conditionals, recursion, loops, and simple data structures. We follow this with the Java Collections framework, generics, object-oriented programming, exceptions, I/O, searching and sorting algorithms, algorithm analysis, and modern Java features such as pattern matching and concurrency.

The ordering of topics presented in a Java course is hotly-debated and has been ever since its creation and use in higher education. Such questions include the location of object-oriented programming principles: do we start off with objects or hold off until later? Depending on the style of a text, either option can work. Even though we, personally, are more of a fan of the "early objects" approach, and is how we learned Java many moons ago, we choose to place objects later in the curriculum. We do this to place greater emphasis on testing, method design, recursion, and data structures through the Collections framework. Accordingly, after our midterm (roughly halfway through the semester), students should have a strong foundation of basic Java syntax sans objects and class design. The second half of the

class is dedicated to just that: object-oriented programming and clearing up confusions that coincide and introduce themselves.

We believe that this textbook can be used as any standard second-semester computer science course. At the high-school level, this book can be used as a substitute for the College Board's former AP Computer Science AB course. Certainly, this text may be well-suited for an AP Computer Science A course at the high-school level, but its material goes well beyond the scope of the AP exam and curriculum. Instructors are free to omit certain topics that may have been covered in a prerequisite (traditionally-styled) Java course. In those circumstances, it may be beneficial to dive further into the chapters on algorithm analysis and modern Java pragmatics. For students without a Java background (or instructors of said students), which we assume, we take the time to quickly yet effectively build confidence in Java's quirky syntax. Additionally, we understand that our approach to teaching loops (through recursion and a translation pipeline) may appear odd to some long-time programmers. So, an instructor may reorder these sections however they choose, but we strongly recommend retaining our chosen ordering for pedagogical purposes, particularly for those readers that are not taking a college class using this text.

At the end of Chapters 1–5, we present a slew of exercises to the readers, and we encourage them to do most, if not all, of the exercises. Some tasks in the exercises serve as simple reinforcement of topics, whereas others are long-haul marathons that take many hours to complete. Several exercises have also been used as (written) exam questions. We do not provide answers to the exercises because there are many "avenues to success." Readers should collaborate with others to solve the problems and discuss difficulties and points of confusion in aims of clarification.

The diction of our book is chosen very carefully, with hourly of scrutiny dedicated to the paragraphs, sentence structure, and their accompanying presentation. When we demonstrate an example, stop, and closely follow along. Do not rush through it. The words on the text remain in place, no matter the pace of the reader. A page, example, exercise, or chapter may require multiple passes to fully digest the content. Make notes in the margins, type out any examples and exercises, ask questions, and answer the questions posed by others. Perhaps they may answer one of yours.

Once again, by writing this book, we wish to ensure that students are better prepared for the more complex courses in a common computer science curriculum, e.g., data structures, operating systems, algorithms, programming languages, and whatever else lies ahead. A strong foundation keeps students motivated and pushes them to continue even when times are arduous, which we understand there to be plenty thereof. After all, not many moons ago were we in the shoes of our target audience!

Have a blast!
Joshua Crotts

Acknowledgements

I piloted *Learning Java* on the students in the Fall 2023 and Spring 2024 semesters of CSCI-C 212 (Introduction to Software Systems) at Indiana University. The inspiration came from those in the former, which drove me to continue writing and designing exercises. I sincerely appreciate all of the comments, suggestions, and corrections made. The following students and course staff members found mistakes: Ashley Won, Daniel Yang, Jack Liang, Shyam Makwana, and Muazzam Siddiqui.

Contents

A JUnit Testing

Welcome to the back of the book; we hope this is not after you have finished the book but rather before you have even started the main content! In this Appendix, we will discuss how to use and setup the JUnit testing framework.

A.1 JUnit

There are many testing frameworks that we could use during our Java adventure, but we will stick to the industry-standard JUnit library. JUnit allows us to write test cases for methods as a means of determining whether they function correctly. Most beginning programmers debug or test their methods by calling them in, for example, the `main` method with inputs, then verifying that their output matches what they expect, usually through the console. This is neither robust nor elegant, and is prone to mistakes. Moreover, it introduces an unnecessary step: having to check to see whether the terminal contains the correct output. JUnit bypasses this inconvenience and we will demonstrate with some examples.

A.1.1 Installing & Using JUnit

Firstly, we need to install JUnit. We will assume that the users are working with the IntelliJ IDE and have it installed on their computer. Each project has to have JUnit separately configured, but doing so is trivial. There are two primary ways of integrating JUnit into a project: with or without Maven, which is a complex dependency manager. Our examples do not use Maven.

We need to create a class definition that has our method to test. Let's redo the example from Chapter **??**, where we convert a temperature from degrees Fahrenheit to degrees Celsius.

```
class TempConverter {

  /**
   * Converts a temperature from Fahrenheit to Celsius.
   * @param f - degrees Fahrenheit.
   * @return f in degrees Celsius.
   */
  static double fToC(double f) {
    return 0.0;
  }
}
```

(1) In IntelliJ, right-click the class name, i.e., `TempConverter`, then click "Show Context Actions" (you can also press a shortcut combination such as Alt+Enter). This pops up a menu with a few options, one of which is "Create test."
(2) Click this option and a dialog box will appear labeled "No Test Roots Found," which asks if you want to create the tests in the same directory as the source files. In large projects, it is a good idea to separate the source files from tester files, but for our purposes, they will remain together.
(3) Click "OK," and another dialog box should appear, containing various options, the first of which is a piece of text saying that "JUnit5 library not found in the module." Beside of this text is a button labeled "Fix".
(4) Click "Fix", then hit "OK" in the following dialog box.
(5) Now, at the bottom, there is a box with all the visible methods for which we can write tests. In our case, the only option is `fToC`, which is to be expected. Click the checkbox to its left, the hit "OK" to generate the test file.
(6) From here, IntelliJ generates a new and separate class/source called `TempConverterTest`. Assuming everything is working up to this point, there are two pieces of red text, one of which is "Assertions" on line one and the other is "Test" on line five.
(7) Hover your cursor over the "Assertions" word and wait for about two seconds. A tooltip should appear saying that it "Cannot resolve symbol 'Assertions'." Below this is a button that says "Add library JUnit 5.X.Y to classpath," where X and Y are arbitrary versions of JUnit (as long as it is not JUnit 4).
(8) Click "Add library JUnit 5.X.Y to classpath" bring JUnit 5 into the project and the other error should disappear.
(9) Inside the tester file is the `fToC` method with a preluding *annotation*. This `@Test` annotation tells JUnit that this method contains JUnit tests and should be interpreted as such.[1]

Let's write a few tests! To do so, we can use the `assertEquals` method, which receives two arguments: the expected value and the actual value. The expected value, as its name might imply, is the expected output of a method that we test. The actual value, on the other hand, is where we call the method we are testing. So, we might write a test case saying that 212°F is equal to 100°C and another test to assert that 32°F is equal to 0°C. To emphasize that we are working inside a test method, we will prefix `fToC` with `test`, which also helps to eliminate accidentally referencing the `fToC` method defined in this file versus the one inside the `TempConverter` class.

```
import static org.junit.jupiter.api.Assertions.*;

class TempConverterTest {

  @org.junit.jupiter.api.Test
  void testfToC() {
    assertEquals(0, TempConverter.fToC(32));
    assertEquals(100, TempConverter.fToC(212));
  }
}
```

To execute this test (and only this test) method, click the green arrow immediately to the left of the method declaration. This will run the tests inside the method. At the bottom of

[1] Initially, your annotation will contain more than just `@Test`; IntelliJ qualifies the annotation with its full location in the Java library.

the IDE, any failed assertions will be displayed. Of course, the second one fails because we have no meaningful implementation of `fToC` yet.

Should we want to write multiple test methods for different methods in the source file, we can do so easily.

(1) Head back to the `TempConverter.java` file and write the `c2f` method, which converts a temperature in degrees Celsius to degrees Fahrenheit.
(2) Then, click on the class name, "Show Context Actions", and "Create Test."
(3) The same dialog box with the selectable methods appears, so be sure to check `c2f`.
(4) Hitting "OK" at this point displays an error dialog box sayhing that the test file already exists. This is correct and IntelliJ is making sure that we are okay with updating the contents of that file by introducing a new method stub for testing `c2f`.
(5) Hit "OK" and you will see that the `c2f` method now has a corresponding tester method.

Writing assertions in this method is similarly straightforward, and if we do not want to rerun the tests for `fToC` just yet, we do not have to; clicking on the arrows beside a tester method's signature runs only the assertions inside that particular testing method. Should we want to run all the tests, we do not need to click each arrow, as that would be cumbersome. Instead, at the class declaration, i.e., `class TempConverterTest`, there is another green arrow; clicking this button runs all declared test methods inside our class definition.

One issue that arises when running tests with `assertEquals` and other variants is that, if an assertion fails, JUnit stops execution at the point of failure and refuses to run subsequent tests. This is not very convenient, so to circumvent the issue, we wrap all assertion statements inside a call to `assertAll`, which acts as a "dispatch" of sorts. We provide, to `assertAll`, a list of assertions to execute, and it executes them one after another, regardless of if one fails. A syntax warning to be aware of is that each assertion must be prefaced with '`() -> `', without the quotes, and all but the last assertion must have commas.[2] Below is an example.

```
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class TempConverterTest {

  @Test
  void fToC() {
    assertAll(
      () -> assertEquals(0, TempConverter.fToC(32)),
      () -> assertEquals(100, TempConverter.fToC(212)),
      () -> assertEquals(-40, TempConverter.fToC(-40)));
  }
}
```

Rerunning this test demonstrates that, even though the second assertion fails, the last is still executed because all of the assertions reside within a call to `assertAll`.

Figure A.1 provides a table of helpful JUnit assertion methods.

[2] The significance of `() ->` is unimportant for now.

JUnit 5 Testing Methods

`assertEquals(`e`, `a`)` asserts that the actual value, namely a, should be equal to the expected value e. When these are primitive datatypes, e.g., `int`, their values are compared. If they are objects, it uses their `equals` method implementation.

`assertNotEquals(`e`, `a`)` is the dual to `assertEquals` in that, if `assertEquals(`e`, `a`)` returns `true`, then `assertNotEquals(`e`, `a`)` returns false.

`assertTrue(`p`)` asserts that p is an expression that resolves to `true`.

`assertFalse(`p`)` asserts that p is an expression that resolves to `false`.

`assertArrayEquals(`A_2`, `A_1`)` asserts that the contents of an expression generating the array A_1 are equal to the expected array of values A_2.

`assertThrows(`E`, `e`)` asserts that the executable code e throws the exception E.

`assertNull(`e`)` asserts that e is `null`.

Fig. A.1: Useful JUnit Methods.

B Binary Search Tree Library

In this appendix, we provide the implementation of the binary search tree library used in the book, because Java does not provide a built-in collection dedicated solely to binary search trees. On the other hand, some of its data structures, such as `TreeSet` and `TreeMap`, are implemented using trees, but the underlying details are abstracted from the users.

B.1 Binary Search Tree Library

References