Teaching Java
A Test-Driven Approach

Joshua Crotts

January 17, 2024

Department of Computer Science
Indiana University Bloomington

Teaching Java: A Test-Driven Approach

Joshua Crotts

Contents

List of Figures

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 should 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. Immediately diving into a method implementation often results in endless hours of debugging that could have been saved by 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, 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 then follow this with the Java Collections API, generics, class design, advanced object-oriented programming, 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. While 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 API. 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. 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 in whatever order 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.

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 to be plenty thereof.

Have a blast!
Joshua Crotts

1. Testing & Java Basics

1.1 A First Glimpse at Java

It makes little sense to avoid the topic at hand, so let us jump right in and write a program! We have seen *functions* before, as well as some mathematics operations, perhaps in a different (language) context.

Example 1.1. Our program will convert a given temperature in Fahrenheit to Celsius.

Listing 1.1

```
class TempConverter {

  /**
   * Converts a temperature from Fahrenheit to Celsius.
   * @param temperature in F.
   * @return temperature in C.
   */
  static double fahrenheitToCelsius(double d) {
    return 0.0;
  }
}
```

All code, in Java, belongs to a *class*. Classes have much more complex and concrete definitions that we will investigate in due time, but for now, we may think of them as the homes of our functions. By the way, functions in Java are called *methods*.[1] Again, this slight terminology differentiation is not without its reasons, but for all intents and purposes, functions are methods and vice versa. The class we have defined in the previous listing is called `TempConverter`, giving rise to believe that the class does something related to temperature conversion.

We write the `fahrenheitToCelsius` method, whose *return type* is a `double`, and has one *parameter*, which is also a `double`. A `double` is a floating-point value, meaning it potentially

[1] The reasoning is simple: a method belongs to a class. Other programming languages, e.g., C++ or Python, do not restrict the programmer to writing code only within a class. Thus, there is a distinction between functions, which do not reside within a class, and methods.

has decimals. For our method, this choice makes sense, because if we were to instead receive an `int`, we would not be able to convert temperatures such as `35.5` degrees Fahrenheit to Celsius.

The `static` keyword that we wrote has significance, but for now, consider it a series of six mandatory key presses (plus one for the space thereafter).

Above this method *signature* is a Java documentation comment, providing a brief summary of the method's purpose, as well as the data it receives as parameters and its return value, should it be necessary.

Inside its method body lies a single `return`, in which we return `0.0`. Returning a value is what a *method call*, or *method invocation*, resolves to. For example, if we were to call `fahrenheitToCelsius` with any arbitrary `double` value, the call would be substituted with `0.0`. This is otherwise called *method application*.

```
fahrenheitToCelsius(5)      -> 0.0
fahrenheitToCelsius(78)     -> 0.0
fahrenheitToCelsius(-3123) -> 0.0
```

Of course, this method is meaningless without an implementation. We want to design *test cases* to ensure the method works as expected. Test cases verify the correctness (or incorrectness) of a method. We, as the readers, know how to convert a temperature from Fahrenheit to Celsius, but telling a computer to do such a conversion is not as obvious at first glance. To test our methods, we will use the *JUnit* testing framework. To create a test for `fahrenheitToCelsius`, we will make a second class called `TempConverterTester` to house a single method: `fahrenheitToCelsiusTest`.

Listing 1.2

```
class TempConverterTester {

  @Test
  void testFahrenheitToCelsius() {
    Assertions.assertAll(
      () -> Assertions.assertEquals(0, TempConverter.fahrenheitToCelsius(32)));
  }
}
```

We want JUnit to recognize that `fahrenheitToCelsius` contains testing code, so we prepend the `@Test` annotation to the method signature. In its body, we call `Assertions.assertAll`, which receives a series of methods that are ran in succession. In our case, we want to assert that our `fahrenheitToCelsius` method should return `0` degrees Celsius when given a temperature of `32` degrees Fahrenheit. The first parameter to `assertEquals` is the expected value of the test, i.e., what we want it to produce. The second parameter is the actual value of the test, i.e., what our code produces.

When writing tests, it is important to consider *edge cases* and all possible branches of a method implementation. Edge cases are inputs that are possibly missed by an implementation, e.g., `-40`, since it is the same in both Fahrenheit and Celsius, or `0`. So, let us add a few more test cases.[1]

[1]To condense our code, we use a `static` import of our `fahrenheitToCelsius` method, which allows us to call the method without specifying the full class name. We apply this same idea to those methods used from the `Assertions` class.

Listing 1.3

```
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static TempConverter.fahrenheitToCelsius;

class TempConverterTester {

  @Test
  void testFahrenheitToCelsius() {
    assertAll(
      () -> assertEquals(0, fahrenheitToCelsius(32)),
      () -> assertEquals(100, fahrenheitToCelsius(212)),
      () -> assertEquals(-40, fahrenheitToCelsius(40)),
      () -> assertEquals(-17.778, fahrenheitToCelsius(0), .01),
      () -> assertEquals(-273.15, fahrenheitToCelsius(-459), .01));
  }
}
```

Floating-point arithmetic can cause precision/rounding errors. So, as an optional third argument to `assertEquals`, we might provide a *delta*, which allows for precision up to a certain amount to be accepted as a valid answer. For example, our tolerance for the fourth and fifth test cases is `0.01`, meaning that if our actual value is less than $\pm$ 0.01 away from the expected value, the test case succeeds.

Now that we have copious amounts of tests, we can write our method definition. Of course, it is trivial to write.

Listing 1.4

```
class TempConverter {

  /**
   * Converts a temperature from Fahrenheit to Celsius.
   * @param temperature in F.
   * @return temperature in C.
   */
  static double fahrenheitToCelsius(double d) {
    return (d - 32) * (5.0 / 9.0)
  }
}
```

This definition brings up a few points about Java's type system. The primitive mathematics operations account for the types of its arguments. So, for instance, subtracting two integers will produce another integer. More noteworthy, perhaps, a division of two integers produces another integer, even if that result seems to be incorrect. Thus, `5 / 9` results in the integer `0`. If, however, we treat at least one of the operands as a floating-point value, we receive a correct result of approximately `0.555555`: `5.0 / 9`. Java by default uses the standard order of operations when evaluating mathematics expressions, so we force certain operations to occur first via parentheses.[1]

[1] By "standard", we mean the widely-accepted paradigm of parentheses first, then exponents, followed by left-to-right multiplication/division, and finally left-to-right addition/subtraction.

Unlike some programming languages that are *dynamically-typed*, e.g., Scheme, Python, JavaScript, the Java programming language requires the programmer to specify the types of variables.[1] Java has several default *primitive datatypes*, which are the simplest reducible form of a variable. Such types include `int`, `char`, `double`, `boolean`, and others. Integers, or `int`, are any positive or negative number without decimals. Doubles, or `double`, are values with decimals. Characters, or `char`, are a single character enclosed by single quotes, e.g., `'X'`. Finally, booleans, or `boolean`, are either `true` or `false`. There are other Java data types that specify varying levels of precision for given values. Integers are 32-bit signed values, meaning they have a range of $[-2^{31}, 2^{31})$. The `short` data type, on the contrary, is 16-bit signed. Beyond this is the `byte` data type that, as its name suggests, stores 8-bit signed integers. Floating-point values are more tricky, but while `double` uses 64 bits of precision, the `float` data type uses 32 bits of precision.

Example 1.2. Let us write a method that receives two three-dimensional vectors and returns the distance between the two. We can, effectively, think of this as the distance between two points in a three-dimensional plane. Therefore because each vector contains three components, we need six parameters, where each triplet represents the vectors v_1 and v_2.

Listing 1.5

```
class VectorDistance {

  /**
   * Computes the distance between two given vectors:
   * v_1=(x_1, y_1, z_1) and v_2=(x_2, y_2, z_2)
   */
  static double computeDistance(double x1, double y1, double z1,
                                double x2, double y2, double z2) {
    return 0.0;
  }
}
```

Again we start by writing the appropriate method signature with its respective parameters and a Java documentation comment explaining its purpose. For tests, we know that the distance between two Cartesian points in a three-dimensional plane is

$$D(v_1, v_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

So, let's now write a few test cases with a few arbitrarily-chosen points that we can verify with a calculator or manual computation.

Listing 1.6

```
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static VectorDistance.computeDistance;

class VectorDistanceTester {

  @Test
  void testComputeDistance() {
    assertAll(
      assertEquals(8.66, computeDistance(3, 2, 1, 8, 7, 6), .01),
```

[1] In Java 10, the `var` keyword was introduced, which automatically infers the type of a given expression.

```
      assertEquals(12.20, computeDistance(0, 0, 0, 8, 7, 6), .01),
      assertEquals(8.30, computeDistance(-8, -2, 1, 0, 0, 0), .01));
  }
}
```

Notice again our use of the optional delta parameter to allow us a bit of leeway with the rounding of our answer. Fortunately, the implementation of our method is just a retelling of the mathematical definition.

Listing 1.7

```
class VectorDistance {

  /**
   * Computes the distance between two given vectors:
   * v_1=(x_1, y_1, z_1) and v_2=(x_2, y_2, z_2)
   */
  static double computeDistance(double x1, double y1, double z1,
                                double x2, double y2, double z2) {
    return Math.sqrt(Math.pow(x1 - x2, 2)
                   + Math.pow(y1 - y2, 2)
                   + Math.pow(z1 - z2, 2));
  }
}
```

We make prolific use of Java's `Math` library in designing this method; we use the `sqrt` method for computing the square root of our result, as well as `pow` to square each intermediate difference.

Example 1.3. Slope-intercept is an incredibly common algebra and geometry problem, and even pokes its way into machine learning at times when computing best-fit lines. Let's write two methods, both of which receive two points (x_1, y_1), (x_2, y_2). The first method returns the slope m of the points, and the second returns the y-intercept b of the line. Their respective signatures are straightforward—each set of points is represented by two integer values and return doubles.

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$
$$b = y_1 - mx_1$$

Listing 1.8—Computing the Slope Intercept Method Signatures

```
class SlopeIntercept {

  /**
   * Computes the slope of the line represented by the two Cartesian points.
   * @return slope of points.
   */
  static double slope(int x1, int y1, int x2, int y2) {
    return 0.0;
  }

  /**
   * Computes the y-intercept of the line represented by the two Cartesian points.
   * @return y-intercept of line represented by points.
   */
  static double yIntercept(x1, int y1, int x2, int y2) {
    return 0.0;
```

```
  }
}
```

The tests that we write are verifiable by a calculator or mental math. Note that the `yIntercept` method depends on a successful implementation of `slope`, as designated by the formula of the former. In the next chapter, we will consider cases that invalidate the formula, e.g., when two points share an x coordinate, in which the slope is undefined for those points.

Listing 1.9—Testing the Slope Intercept Methods

```
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static SlopeIntercept.slope;
import static SlopeIntercept.yIntercept;

class SlopeInterceptTester {

  @Test
  void testSlope() {
    assertAll(
      () -> assertEquals(1, slope(0, 0, 1, 1)),
      () -> assertEquals(0, slope(0, 0, 1, 0)),
      () -> assertEquals(2, slope(8, 4, 2, 4)),
      () -> assertEquals(0.5, slope(-1, 5, 3, 7)));
  }

  @Test
  void testYIntercept() {
    assertAll(
      () -> assertEquals(0, yIntercept(0, 0, 1, 1)),
      () -> assertEquals(0, yIntercept(0, 0, 1, 0)),
      () -> assertEquals(4, yIntercept(8, 4, 2, 4)),
      () -> assertEquals(5.5, yIntercept(-1, 5, 3, 7)));
  }
}
```

And the implementation of the two methods follows from the mathematical definitions. We replace our temporary `0.0` return values with the appropriate expressions, and all tests pass.

Listing 1.10—Implementing the Slope Intercept Methods

```
class SlopeIntercept {

  /**
   * Computes the slope of the line represented by the two Cartesian points.
   * @return slope of points.
   */
  static double slope(int x1, int y1, int x2, int y2) {
    return (y2 - y1) / (x2 - x1);
  }

  /**
   * Computes the y-intercept of the line represented by the two Cartesian points.
   * @return y-intercept of points.
   */
  static double yIntercept(x1, int y1, int x2, int y2) {
    return y1 - slope(x1, y1, x2, y2) * x1;
  }
```

```
}
```

Example 1.4. We are starting to get used to some of Java's verbosity! Let us now write a method that, when given a value of x, evaluates the following quartic formula:

$$q(x) = 4x^4 + 7x^3 + 21x^2 - 65x + 3$$

Its signature is straightforward: we receive a value of x, namely a `double`, and return a `double` since we are performing mathematical operations on `double` values. Again, we return zero as a temporary solution to ensure the program successfully compiles.

Listing 1.11

```
class QuarticFormulaSolver {

  /**
   * Evaluates the following quartic equation:
   * 4x^4 + 7x^3 + 21x^2 - 65x + 3.
   * @param the input variable x.
   * @return the result of the expression after substituting x.
   */
  static double solveQuartic(double x) {
    return 0.0;
  }
}
```

Test cases are certainly warranted, but may be a bit tedious to compute by hand, so we recommend using a verified calculator to compute expected solutions![1]

Listing 1.12

```
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static QuarticFormulaSolver.solveQuartic;

class QuarticFormulaSolverTester {

  @Test
  void testSolveQuartic() {
    assertAll(
      () -> assertEquals(3, solveQuartic(0)),
      () -> assertEquals(510, solveQuartic(3)),
      () -> assertEquals(229878, solveQuartic(15)),
      () -> assertEquals(313445.1875, solveQuartic(16.25)));
  }
}
```

Again, we write tests *before* the method implementation because we know, intuitively, how to solve an equation for a variable, but a computer has to be told how to solve this task. Fortunately for us, a quartic equation solver is nothing more than returning the result of the expression. We, of course, have to use the exponential `Math.pow` method again (or conjoin several multiplicatives of x), but otherwise, it is straightforward.

[1] Remember that the coefficient is applied *after* applying the exponent to the variable. That is, if $x = 3$, then $4x^3$ is equal to $4 \cdot (3)^3$, which resolves to $4 \cdot 27$, which resolves to 108.

Listing 1.13

```
class QuarticFormulaSolver {

  /**
   * Evaluates the following quartic equation:
   * 4x^4 + 7x^3 + 21x^2 - 65x + 3.
   * @param the input variable x.
   * @return the result of the expression after substituting x.
   */
  static double solveQuartic(double x) {
    return 4 * Math.pow(x, 4) + 7 * Math.pow(x, 3) + 21 * Math.pow(x, 2) - 65 * x + 3;
  }
}
```

And, as expected, all tests pass. With only methods and math operations at our disposal, the capabilities of said methods is quite limited. Let us start revamping our tool set by reintroducing strings.

1.2 Strings

Strings, as you might recall, are a sequence of characters. Characters, of course, are enclosed by single quotes, e.g., `'x'`. A Java `String` is enclosed by double quotes. e.g., `"Hello!"`. Strings may contain any number of characters and any kind of character, including no characters at all or only a single character. There is an apparent distinction between `'x'` and `"x"`: the former is a `char` and the latter is a `String`! Note the capitalization on the word `String`; this is a significant detail, because a `String` is not one of the primitive datatypes that we described in the previous section. Instead, it is several `char` values combined together "under-the-hood", so to speak. We can declare a `String` as a variable using the keyword combined with a variable name, just as we might for primitives.

Listing 1.14

```
class NewStringTests {

  public static void main(String[] args) {
    String s1 = "Hello, world!";
    String s2 = "How are you doing?";
    String s3 = "This is another string!";
  }
}
```

We can conjoin, or *concatenate*, strings together with the + operator. Concatenating one string s_2 onto the end of another string s_1 creates a new string s_3, copies the characters from s_1 as well as the characters from s_2, in that order.

To retrieve the number of characters in a string, invoke `.length` on the string. Note that the empty string has length zero, and spaces count towards the length of a string, since spaces are characters. For instance, the length of `" a "` is five because there are two spaces, followed by a lowercase 'a', followed by two more spaces.

Comparing strings for equality seems straightforward: we can use == to compare one string versus another. Doing this is a common beginner pitfall! Strings are *objects* and cannot be

compared for value-equality using the == operator. Introducing this term "value-equality" insinuates that strings can, in fact, be compared using ==, which is theoretically correct. The problem is that the result of this comparison only compares the *hashcodes* of the strings. In other words, s_1 == s_2 returns whether the two strings reference the same string in memory. All objects, strings included, have a hashcode, which (very) roughly corresponds to a location in memory; it provides an identifier for an object. An object o_1 that shares the same hashcode as another object o_2 implies that o_1 and o_2 are definitionally equivalent, meaning they represent the same object.

Listing 1.15

```
class NewStringTests {

  public static void main(String[] args) {
    String s1 = "Hello";
    String s2 = s1;
    System.out.println(s1 == s2); // true
  }
}
```

In the above code snippet we declare s_1 as the *string literal* `"Hello"`, then initialize s_2 to point to s_1. So there are, in effect, two pointers to the string literal `"Hello"`. Let us try something a little more tricky: suppose we declare three strings, where s_1 is the same as before, s_2 is the string literal `"Hello"`, and s_3 is the string literal `"World"`. Comparing s_1 to s_2, strangely enough, also outputs true, but why? It seems that s_1 and s_2 reference different string literals, even though they contain the same characters. Indeed, this is the case, but Java performs an optimization called *string pool caching*. That is, if two strings *are* the same string literal, it makes little sense for them to point to two distinct hashcodes, since strings are immutable. Therefore, Java optimizes these into references to a single allocated string literal. Comparing s_1 or s_2 with s_3 outputs false, which is the anticipated result.

Listing 1.16

```
class NewStringTests {

  public static void main(String[] args) {
    String s1 = "Hello";
    String s2 = "Hello";
    String s3 = "World";
    System.out.println(s1 == s2); // true (?)
    System.out.println(s2 == s3); // false
  }
}
```

If we want to circumvent Java's string caching capabilities, we need a way of *instantiating* a new string for our variables to reference. We use the power of `new String` to create a brand new, non-cached string reference. We treat this as a method, of sorts, called the *object constructor*. We will revisit constructors in our discussion on objects and classes, but for now, consider it a method for creating a distinct `String` instance. This method is *overloaded* to receive either zero or one parameter. The latter implementation receives a `String`, which is copied into the new `String` instance. If we pass a string literal to the constructor, it copies the characters *from*

the literal into the new string. At this point, the only possible object to be equal to s_1 is s_1 itself or another string that points to its value.

Listing 1.17

```
class NewStringTests {

  public static void main(String[] args) {
    String s1 = new String("hello");
    String s2 = new String("hello");
    String s3 = new String("world");
    String s4 = s1;
    System.out.println(s1 == s1); // true
    System.out.println(s1 == s2); // false!
    System.out.println(s2 == s3); // false
    System.out.println(s1 == s4); // true
  }
}
```

Now, what if we want to compare strings for their content, i.e., their characters? The `String` class provides a handy `.equals` method that we invoke on instances of strings. Two strings s_1 and s_2 are equal if they are *lexicographically equal*. In essence, this is a long and scary word to represent the concept of "containing the same characters". Lexicographical comparisons are case-sensitive, meaning that uppercase and lowercase letters are different according to Java.

Listing 1.18

```
class StringLexicographicallyEqual {

  public static void main(String[] args) {
    String s1 = new String("hello");
    String s2 = new String("hello");
    String s3 = new String("world");
    System.out.println(s1.equals(s2)); // true
    System.out.println(s2.equals(s3)); // false
  }
}
```

Let us veer into the discussion on the lexicographical ordering of strings. Like numbers, strings are comparable and can be, e.g., "less than" another. According to Java, one string is less than another if it is lexicographically less than another, and this idea extends to all comparison operators aside from equality. Memorizing the string ordering rules is cumbersome, so we propose the S.N.U.L. acronym. In general, special characters (S) are less than numbers (N), which are less than uppercase characters (U), which are less than lowercase characters (L). For a full description with the exceptions to our generalization, view an ASCII table.

Example 1.5. Java returns the distance between the first non-equal characters in a string using `.compareTo`. For instance, `"hello".compareTo("hi")` returns `-4` because the distance from the first non-equal characters, those being `'e'` and `'i'`, is minus four characters, since `'i'` is greater than `'e'`. If we compare `"hello"` against `"Hello"`, we get `32`, because according to the ASCII table, `'h'` corresponds to the integer 104, and `'H'` corresponds to the integer 72, and their difference is $104 - 72$. Of course, comparing `"hello"` against `"hello"` produces zero, indicating that they contain the same characters.

Strings are *indexed from zero*, which means that the characters in the string are located at *indices* from zero to the length of the string minus one. So, for example, in the string `"hello"`, `'h'` is at index zero, `'e'` is at index one, `'l'` is at index two, `'l'` is at index three, and `'o'` is at index four. Knowing this fact is crucial to working with helper methods such as `charAt`, `indexOf`, and `substring`, which we will now discuss.

Example 1.6. To retrieve the character at a given index, we invoke `charAt` on the string: `"hello".charAt(1)` returns `'e'`. Attempting to index out of bounds with either a negative number or a number that is equal to or exceeds the length of the string results in a `StringIndexOutOfBoundsException` error.

Example 1.7. We can use `indexOf` to find the index of the first occurrence of some value in a string. To demonstrate, `"hello, how are you?".indexOf("are")` returns `11` because the substring `"are"` occurs starting at index 11 of the provided string. If the supplied value is not present in the string, `indexOf` returns -1.

Example 1.8. A *substring* of a string s is a sequence of existing characters inside s. We can extract a substring from some string s using the `substring` method: `"abcde".substring(2, 4)` returns `"cd"`. Note that the last index is exclusive, meaning that, as a rule of thumb, the number of characters returned in the substring is equal to the second argument minus the first argument. There is also a handy second version of this method that receives only one argument: it returns the substring from the given index up to the end of the string. E.g., `"abcdefg".substring(1)` returns everything but the first character, i.e., `"bcdefg"`.

Example 1.9. We can convert between datatypes, such as an integer to a string and vice versa, using the `String.valueOf` and `Integer.parseInt` methods respectively. Passing any non-string primitive value as an argument to `valueOf` converts it into a string. Oppositely, if we pass a string that represents an integer to `parseInt`, we obtain the corresponding integer. The `Double.parseDouble` and `Boolean.parseBoolean` methods behave similarly. Passing an invalid string, i.e., one that does not represent the respective datatype, results in Java throwing a `NumberFormatException` error.

```
String s1 = String.valueOf(1234);
String s2 = String.valueOf(Math.PI);
int n1 = Integer.parseInt(s1);
double s2 = Double.parseDouble(s2);
```

1.3 Standard I/O

Early on in a Java programmer's career, they encounter the issue of reading from the "console", or standard input, as well as the dubiously useful act of debugging by printing data to standard output. Many programmers are aptly familiar with these when coming from other programming languages.

First, we need to discuss the nature of the *standard data streams*. Java (and the operating system in general), utilizes three standard data streams: standard input, standard output, and standard error. We can think of these as sources for reading data from and writing data to. The *standard output stream* is often accessed using the `System.out` class, then through its various methods, e.g., `println`, `print`, and `printf`. To output a line to standard output, we invoke `System.out.println` with a string (or some other datatype that is coerced into a string).

String Class

A *string* is an immutable sequence of characters. Strings are indexed from 0 to $|S|-1$, where $|S|$ is the number of characters of S.

S_1 `+` S_2 adds the characters from S_2 onto the end of S_1, producing a new string.

`int` S`.length()` returns the number of characters in S.

`char` S`.charAt(`i`)` retrieves the $(i+1)^{\text{th}}$ character in S. We can also say that this retrieves the character at index i of S.

`String` S`.substring(`i`,` j`)` returns a new string containing the characters from index i, inclusive, to index j, exclusive. The number of extracted characters is $j-i$. We will use the notation $S' \sqsubseteq S$ to denote that S' is a substring of S.

`String` S`.substring(`i`)` returns a new string from index i to the end of S.

`int` S`.indexOf(`S'`)` returns the index of the first instance of S' in S, or -1 if $S' \not\sqsubseteq S$.

`boolean` S`.contains(`S'`)` returns `true` if $S' \sqsubseteq S$; `false` otherwise.

`String` S`.repeat(`n`)` returns a new string containing n copies of S.

`String String.valueOf(`v`)` **returns a stringified version of v, where v is some primitive value.**

`int Integer.parseInt(`S`)` **returns the integer representation of a string S, if it can be parsed as such.**

Figure 1.1: Useful String Methods.

For relaying messages to the user in a terminal-based application or even when debugging a program, outputting information to standard output is a good idea. On the other hand, sometimes a program fails or the programmer wants to output an error message. While it is possible to output error messages to standard output, since they are otherwise indistinguishable, Java has a dedicated *standard error stream* for outputting error messages and logs via `System.err`. We glossed over this method, but let's discuss `printf` in more detail due to its inherent power.

The `printf` method originates from C, but is handy for printing multiple values at once without resorting to unnecessary string concatenation. In addition, it preserves the formatting of the data, which is handy when wanting to treat a `double` as a floating-point number in a string representation. It receives at least one argument: a format string, and is one of several *variadic methods* that we will discuss. A format string contains special format characters and possibly other text.

Example 1.10. To output an `int` or `long` using `printf`, we use the `%d` format specifier.

```
int x = 42;
System.out.printf("The value of x is %d\n", x);
System.out.printf("We can inline ints 42 or as literals %d\n", 42);
```

Example 1.11. To output a `double` using `printf`, we use the `%f` format specifier. We can also specify the number of digits `n` to print after the decimal point by using the format specifier `.%nf`. Note that floating a decimal to *n* digits does not change the value of that variable; rather, it only changes its string/output representation.

```
double x = 42.0;
System.out.printf("The value of x is %f\n", x);
System.out.printf("PI to 2 decimals is %.2f\n", Math.PI);
System.out.printf("PI with all decimals is %f\n", Math.PI);
```

There are many ways to get creative with `printf`, including space padding, number formatting, left/right-alignment, and more. We will not discuss these in detail, but instead we provide Table **??** of the most common format specifiers.

Format Specifier	Description
`%d`	Integer (`int`/`long`)
`%nf`	Floating-point number to *n* decimals (`float`/`double`)
`%s`	String
`%c`	Character (`char`)
`%b`	Boolean (`boolean`)

Figure 1.2: Common Format Specifiers

The *standard input stream* allows us to "read data from the console". We place this phrase in quotes because the standard input stream is not necessarily the console/terminal; it simply refers to reading characters from the keyboard that are then stored inside this data stream.

Example 1.12. Suppose we want to read an integer from the standard input stream. To do so, we first need to instantiate a `Scanner`, which declares a "pipe", so to speak, from which information is read. It is important to state that, while a `Scanner` may read from the standard input stream, it can read from other input streams, e.g., files or network connections. We will explore this

further in subsequent chapters, but for now, let's declare a `Scanner` object to read from standard input.

```
Scanner in = new Scanner(System.in);
```

The `Scanner` class has handy methods for retrieving data from the stream it is scanning (which we will dub the *scannee*). As we said in the example prompt, to read an integer from the scannee, we use `nextInt`, which retrieves and removes the next-available integer from the scannee data stream. Note that the `Scanner` class is line-buffered, meaning that the data will not be processed by the "accessors", e.g., `nextInt`, until there is a new-line character in the input stream. To force a new-line, we press the "Enter"/"Return" key.

```
Scanner in = new Scanner(System.in);
int x = in.nextInt();
System.out.println(x);
```

Running the program and typing in any 32-bit integer feeds it into standard input, then echos it to standard output. Entering any other non-integer value crashes the program with an `InputMismatchException` exception. So, what if we want to read in a `String` from the scannee; would we use `nextString`? Unfortunately, this is not correct. We need to instead use `nextLine`. The `nextLine` method reads a "line" of text, as a string, from the scannee. We define a "line" as all characters until the first occurrence of a new-line. Invoking `nextLine` consumes these characters, including the newline, from the input stream, and stores them into a variable, if requested. It does not, however, store the newline in the variable.

```
Scanner in = new Scanner(System.in);
String line = in.nextLine();
System.out.println(line);
```

Typing in some characters, which may or may not be numbers, followed by a new-line, stores them in the `line` variable, excluding said new-line. Though, what happens if we prompt for an integer *then* a string? The program does something quite strange. We type the integer, hit "Return", and the program terminates as if it did not prompt for a string. This is because of how both `nextInt` and `nextLine` behave: `nextInt` consumes all data up to but excluding an integer from the input stream; ignoring leading whitespaces. So, after consuming the integer, a new-line character remains in the input stream buffer. Then, `nextLine` intends to wait until a newline is in the buffer. Because the input stream buffer presently contains a new-line, it takes everything before the new-line, which comprises the empty string, and consumes both said empty string and the new-line from the buffer. To circumvent this issue, we can insert a call to `nextLine` in between the calls to `nextInt` and `nextLine`, thereby consuming the lone new-line character, clearing the buffer. Notice that we do not put a return value on the left-hand side of this intermediate `nextLine` invocation; this is because such a variable would hold the empty string, which for the purposes of this program is a worthless (variable) assignment.

```
Scanner in = new Scanner(System.in);
int x = in.nextInt();
in.nextLine();
String line = in.nextLine();
```

Example 1.13. Let's reimplement Python's `input` function, which receives a `String` serving as a prompt for the user to enter data. To make it a bit more user-friendly and elegant, we will add a colon and a space after the given prompt. Because we open a `Scanner` that reads from the

standard input stream, there is no need to worry about, say, calling `nextInt` prior to invoking `input`. If, on the other hand, we declared a static global `Scanner` that reads standard input, and we use that to read an integer *and* inside `input`, we would be in trouble. In our case, the possible scanners connected to the standard input stream differ, so this (the integer-input problem) never occurs.

```
static String input(String prompt) {
  Scanner in = new Scanner(System.in);
  System.out.printf("%s: ", prompt);
  return in.nextLine();
}
```

Example 1.14. Suppose we want to write a method that reads three Cartesian points, as integers, from standard input, and computes the area of the triangle that comprises these points. We can type all integers on the same line, as separated by spaces, because `nextInt` only parses the *next* integer delimited by spaces. And, as we said before, `nextInt` skips over existing trailing spaces in the input stream buffer, so those spaces are omitted. From there, we use the formula for computing the area of the triangle from those points.

$$\frac{x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)}{2}$$

Listing 1.19

```
import java.util.Scanner;

class ThreePointArea {

  /**
   * Computes the area of a triangle given three Cartesian points via standard input.
   * @return the area of the triangle.
   */
  static double computeThreePointArea() {
    Scanner in = new Scanner(System.in);
    int x1 = in.nextInt();
    int y1 = in.nextInt();
    int x2 = in.nextInt();
    int y2 = in.nextInt();
    int x3 = in.nextInt();
    int y3 = in.nextInt();
    return (x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1 - y2)) / 2.0;
  }
}
```

We make a note that reading data from a scanner inside a static method that computes some value is not a very good idea; a better solution would be to read the data inside the `main` method, then call `computeThreePointArea` with the six arguments representing each point.

Listing 1.20

```
import java.util.Scanner;

class ThreePointArea {

  /**
   * Computes the area of a triangle given three Cartesian points as parameters.
```

```
    * @return the area of the triangle.
    */
   static double computeThreePointArea(double x1, double y1,
                                       double x2, double y2,
                                       double x3, double y3) {
     return (x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1 - y2)) / 2.0;
   }

   public static void main(String[] args) {
     Scanner in = new Scanner(System.in);
     int x1 = in.nextInt();
     int y1 = in.nextInt();
     ...
     System.out.println(computeThreePointArea(x1, y1, x2, y2, x3, y3));
   }
 }
```

1.4 Randomness

So-called "true" randomness is difficult to implement from a computing standpoint. Thus, for most intents and purposes (i.e., all of those described in this textbook), it is sufficient to use *pseudorandomess* to generate random values. A pseudorandom number generator computes seemingly random values using a deterministic algorithm, which means that the output values from the generator are predictable. Although it might be incredibly difficult to predict values from a pseudorandom number generator, it is theoretically possible, making them insufficient and insecure for cryptographic schemata and algorithms. For writing, say, a word-guessing game that picks a word from a list at random, it is perfectly reasonable to use a pseudorandom number generator.

Well, how do we generate pseudorandom numbers in Java? There are a few methods, and many textbooks opt to use `Math.random`, which we will explain, but our examples will largely constitute use of the `Random` class. Testing methods that rely on randomness is difficult, so our following code snippets do not come with testing suites.

Example 1.15. Using `Random`, let's generate an integer between 0 and 9, inclusive on both bounds. To do so, we first need to instantiate a `Random` object, which we will call `random`. Then, we should invoke `nextInt` on the `random` object with an argument of 10. Passing the argument n to `nextInt` returns an integer $x \in [0, n-1]$.

```
Random random = new Random();
int x = random.nextInt(10); // x in [0, 9]
```

Example 1.16. Imagine we want to generate an integer between -50 and 50, inclusive on both bounds. The idea is to generate an integer between 0 and 100, inclusive, then subtract 50 from the result.

```
Random random = new Random();
int x = random.nextInt(101) - 50; // x in [-50, 50]
```

Example 1.17. When creating a `Random` object, we can pass a *seed* to the constructor, which is an integer that determines the sequence of pseudorandom numbers generated by our `Random` instance. Therefore if we pass the same seed to two `Random` objects, they will generate the same sequence of pseudorandom numbers. If we do not pass a seed to the constructor, then the

`Random` object uses the current time as the seed. In theory we could use a predetermined seed to write JUnit tests for methods that rely on randomness.

Listing 1.21—Testing Two PRNGs with the Same Seed

```
import static Assertions.assertAll;
import static Assertions.assertEquals;

import java.util.Random;

class DualRandomTester {

  @Test
  void dualRandTest() {
    Random r1 = new Random(212);
    Random r2 = new Random(212);
    for (int i = 0; i < 1_000_000_000; i++) {
      assertEquals(r1.nextInt(1_000_000_000), r2.nextInt(1_000_000_000));
    }
  }
}
```

Admittedly, the above test is somewhat useless since it only tests the efficacy of Java's `Random` class rather than code that we wrote ourselves. Regardless, it is interesting to observe the behavior of two random number generators to see that, in reality, pseudo-randomness is, as we stated, nothing more than slightly advanced math.

Example 1.18. Java provides the `random` method from the `Math` class, which receives no arguments. To do anything significant, we must understand how this method works, i.e., what values it can return. The `Math.random()` method returns a random `double` between $[0, 1)$, where the upper-bound is exclusive. So, we could receive results such as `0.391283114421`, `0`, `0.999999999999`, but never exactly one. We can use basic multiplicative offsets to convert this range into what we might want. For example, to generate a random double value between $[0, 10)$, we can multiply the output by ten, e.g., `Math.random() * 10`.

Example 1.19. To generate a random integer between -5 and 15, inclusive, using `Math.random`, we need to do something similar to our `Random` example. First, we multiply the result of `Math.random()` by 21 to generate a floating-point value between $[0, 21)$. Casting this expression to an integer gives us an integer between $[0, 20]$. Finally, subtracting five gets us the desired range.

```
int x = ((int) (Math.random() * 21)) - 5;
```

Chapter Exercises

Exercise 1.1. (⋆)
Design the `double celsiusToFahrenheit(double c)` method, which converts a temperature from Celsius to Fahrenheit.

Exercise 1.2. (⋆)
Design the `double fiToCm(double f, double in)` method, which receives two quantities in feet and inches respectively, and returns the amount in centimeters.

Exercise 1.3. (⋆)
Design the `int combineDigits(int a, int b)` method, which receives two `int` values between 0 and 9, and combines them into a two-digit number.

Exercise 1.4. (⋆)
Design the `double gigameterToLightsecond(double gm)` method, which converts a distance in gigameters to light seconds (i.e., distance light travels in one nanosecond). There are $1,000,000,000$ meters in a gigameter, and light travels $3,000,000$ meters per second.

Exercise 1.5. (⋆)
Design the `double billTotal(double t)` method, which computes the total for a bill. The total is the given subtotal t, plus 6.75% of t for the tax, and 20% of the taxed total for the tip.

Exercise 1.6. (⋆)
Design the `double grocery(int a, int b, int o, int g, int p)` method, which receives five integers representing the number of apples, bananas, oranges, bunches of grapes, and pineapples purchased at a store. Use the following table to compute the total purchase cost in US dollars.

Item	Price Per Item
Apple	\$0.59
Banana	\$0.99
Orange	\$0.45
Bunch of Grapes	\$1.39
Pineapple	\$2.24

Exercise 1.7. (⋆)
Design the `double pointDistance(double px, double py, double qx, double qy)` method, which receives four double values representing two Cartesian coordinates. The method should return the distance between these points.

Exercise 1.8. (⋆)
Design the `int sumOfSquares(int x, int y)` method, which computes and returns the sum of the squares of two integers x and y.

Exercise 1.9. (⋆)
Design the `double octagonArea(double s)` method, which computes the area of an octagon with a given side length s. The formula is

$$A = 2(1 + \sqrt{2})s^2$$

Exercise 1.10. (⋆)
Design the `double pyramidSurfaceArea(double l, double w, double h)` method, which

computes the surface area of a pyramid with a given base length l, base width w, and height h. The formula is

$$A = lw + l\sqrt{\left(\frac{w}{2}\right)^2 + h^2} + w\sqrt{\left(\frac{l}{2}\right)^2 + h^2}$$

Exercise 1.11. (⋆)
Design the `double crazyMath(double x)` method, which receives a value of x and computes the value of the following expression:

$$(1 - e^{-x})^{xe^{-x}} \cdot \frac{x\pi \cdot \cos(4\pi x)}{\log_2 |x| \cdot \log_4 |x| \cdot \ln |x|}$$

Below are some test cases. Hint: when testing this method, you may want to use the `delta` parameter of `assertEquals`!

Listing 1.22

```
> crazyMath(0)
-0.0
> crazyMath(1)
Infinity
> crazyMath(2)
17.429741427952166
> crazyMath(3)
6.778069159471912
> crazyMath(10)
2.4727699557822547
```

Exercise 1.12. (⋆)
Design the `double logBase(double n, double b)` that, when given a number n and a base b, returns $\log_b(n)$. You will need to make use of the change-of-base formula, which we provide below (n is the number to compute the logarithm of, b is the old base, and b' is the new base).

$$\log_b(n) = \frac{\log_{b'}(n)}{\log_{b'}(b)}$$

Exercise 1.13. (⋆)
Design the `String weekday(int d)` method that, when given an integer d from 1 to 7, returns the corresponding day of the week, with 1 corresponding to `"Monday"` and `"Sunday"` corresponding to 7. You **cannot** use any conditionals or data structures to solve this problem. Hint: declare a string containing each day of the week, with spaces to pad the days, and use `indexOf` and `substring`.

Exercise 1.14. (⋆)
Design the `String flStrip(String s)` method that, when given a string, returns a new string with the first and last characters stripped. You may assume that the input string contains at least two characters.

Exercise 1.15. (⋆)
Design the `double stats(double x, double y)` method that receives two `double` parameters, and returns a `String` containing the following information: the sum, product, difference, the average, the maximum, and the minimum. The string should be formatted as follows, where each category is separated by a newline `'\n'` character. Assume that `XX` is a placeholder for the calculated result.

```
"sum=XX
```

```
product=XX
difference=XX
average=XX
max=XX
min=XX"
```

Exercise 1.16. (⋆)
Design the `String userId(String f, String l, int y)` method that computes a user ID based on three given values: a first name, a last name, and a birth year. A user ID is calculated by taking the the first five letters of their last name, the first letter of their first name, and the last two digits of their birth year, and combining the result. Your method should, therefore, receive two `String` parameters and an `int`. Do not convert the year to a `String`. Below are some test cases.

```
userId("Joshua", "Crotts", 1999)     => "CrottJ99"
userId("Katherine", "Johnson", 1918) => "JohnsK18"
userId("Fred", "Fu", 1957)           => "FuF57"
```

Exercise 1.17. (⋆)
Design the `String cutUsername(String email)` method that receives an email address of the form `x@y.z` and returns the username. The username of an email address is `x`.

Exercise 1.18. (⋆)
Design the `String cutDomain(String url)` method that returns the domain name of a website URL of the form `www.x.Z`, where `X` is the domain name and `Z` is the top-level domain.

Exercise 1.19. (⋆)
Design the `int nextClosest(int m, int n)` method that, when given two positive (non-zero) integers m and n, finds the closest positive integer z to m such that z is a multiple of n and $z \leq m$. For example, if $m = 67$ and $n = 15$, then $z = 60$.

JUnit

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.

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 or not 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.

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 will need 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. For example, let's redo the example from Chapter 1, where we convert a temperature from degrees Fahrenheit to degrees Celsius.

Listing .23

```
class TempConverter {

  /**
   * Converts a temperature from Fahrenheit to Celsius.
```

```
    * @param f - degrees Fahrenheit.
    * @return f in degrees Celsius.
    */
   static double f2c(double f) {
     return 0.0;
   }
}
```

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 will pop up a menu with a few options, one of which is "Create test". Click this, and a dialog box will pop up 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. Click "OK", and another dialog box will 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"; click this, then hit "OK" in the following dialog box. Now, at the bottom, there exists a box with all the visible methods for which we can write tests. In our case, the only option is `f2c`, which is to be expected. Click the checkbox to its left, the hit "OK" to generate the test file.

From here, IntelliJ generates a new and separate class/source called `TempConverterTest`. Assuming everything is correct 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. Hover your cursor over the "Assertions" word, and wait for about two seconds. A tooltip should appear saying that it "Cannot resolve symbol À́ssertions". 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). Clicking this will bring JUnit 5 into the project and the other error should disappear.

Inside this tester file is a method `f2c` with a preluding annotation immediately above. 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 `f2c` with `test`, which also helps to eliminate accidentally referencing the `f2c` method defined in this file versus the one inside the `TempConverter` class.

Listing .24

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

class TempConverterTest {

  @org.junit.jupiter.api.Test
  void testF2c() {
    assertEquals(0, TempConverter.f2c(32));
    assertEquals(100, TempConverter.f2c(212));
  }
}
```

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

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, and in the output window at the bottom of the IDE will be a list of the assertions that failed, if any. Of course, the second one fails because we have no meaningful implementation of `f2c` yet. Should we want to write multiple test methods for different methods in the source file, we can do so easily. Head back to the `TempConverter.java` file and write the `c2f` method, which converts a temperature in degrees Celsius to degrees Fahrenheit. Then, click on the class name, show context actions, and create test. The same dialog box with the selectable methods appears, so be sure to check `c2f`. 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`. 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 `f2c` 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 individual arrow, as that would be cumbersome. Instead, at the class declaration, i.e., `class TempConverterTest`, there is another green arrow; clicking this runs all declared test methods inside our class definition.

One issue that comes up 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 any other tests that follow. This is rarely a good idea, so to circumvent this problem, we can wrap all assertion statements inside a call to `assertAll`, which acts as a "dispatch" of sorts. What this entails is, we provide, to `assertAll`, a list of assertions to execute, and it will execute 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. Below is an example.

Listing .25

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

import org.junit.jupiter.api.Test;

class TempConverterTest {

  @Test
  void f2c() {
    assertAll(
      () -> assertEquals(0, TempConverter.f2c(32)),
      () -> assertEquals(100, TempConverter.f2c(212)),
      () -> assertEquals(-40, TempConverter.f2c(-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 3 provides a table of helpful JUnit assertion methods.

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`.

Figure 3: Useful JUnit Methods.

Bibliography

[Aho et al., 2006] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[Bergin et al., 2013] Bergin, J., Stehlik, M., Roberts, J., and Pattis, R. (2013). *Karel J Robot: A Gentle Introduction to the Art of Object-Oriented Programming in Java*. John Wiley & Sons.

[Bloch, 2018] Bloch, J. (2018). *Effective Java*. Addison-Wesley, Boston, MA, 3 edition.

[Cormen et al., 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. The MIT Press, 3rd edition.

[Felleisen et al., 2018] Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. (2018). *How to Design Programs: An Introduction to Programming and Computing*. The MIT Press.

[Kernighan and Ritchie, 1988] Kernighan, B. W. and Ritchie, D. M. (1988). *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition.

[Nystrom, 2021] Nystrom, R. (2021). *Crafting Interpreters*. Genever Benning.

[Pattis, 1995] Pattis, R. E. (1995). *Karel The Robot: A Gentle Introduction to the Art of Programming*. John Wiley & Sons.

[Siek, 2023] Siek, J. G. (2023). *Essentials of Compilation*. MIT Press, London, England.

[van Orman Quine, 1950] van Orman Quine, W. (1950). *Methods of Logic*. Harvard University Press.

[Weiss, 1998] Weiss, M. A. (1998). Data structures and problem solving using java. *SIGACT News*, 29(2):42–49.

Index