Teaching Java
A Test-Driven Approach

Joshua Crotts

February 10, 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. 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 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

```java
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 has decimals. For our method, this choice makes sense, because if we were

[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.

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

```java
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]

Unlike some programming languages that are *dynamically-typed*, e.g., Scheme, Python, JavaScript, the Java programming language requires the programmer to specify the types

[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.

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

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

```
  }
}
```

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

```java
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

```java
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

```java
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

```java
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.

The Main Method

Java programming tutorials are quick to throw a lot of information at the reader/viewer, and our textbook is no exception to this practice. Unfortunately, Java is one of the more verbose programming languages, and to get up and running with a method, we must wrap its definition inside a class. From there we can, of course, begin to write our `static` method. The question, of course, is what does the static keyword mean? For the first few chapters, we will intentionally omit the definition, as it would almost certainly confuse the reader coming from another language. Therefore, for the time being, simply view it as six characters, plus a space, that must be typed in order to write a method to then test with JUnit. Though, imagine a case in which we want to output the result of some value without using a test, as we will do in many examples. Java requires a `main` method in any executable Java class that does not use tests. For the sake of completeness, let's now write the traditional "Hello, world!" program, but in Java using the `main` method and not tests.

Listing 1.14—Example of the Main Method

```
class MainMethod {

  public static void main(String[] args) {
    System.out.println("Hello, world!");
  }
}
```

Yikes, that is a lot of required code to output some text to the console; what does `public` mean, and what are those `[]` brackets after the `String` word? Once again, we will not detail of their importance but view them as more mandatory characters to type when writing a `main` method. The only word we will explain is `void`, which means that the method does not return a value. If the readers are coming from a functional programming language, e.g., Scheme/Racket or OCaml, then it is almost certainly the case that they never worked with methods that did not return a value nor received no arguments. The `println` method in particular has no return value; its significance is the fact that it outputs some text to the terminal/console, which is a side-effect of the method. We'll come back to what all of this means in subsequent chapters, but we could not avoid at least briefly discussing it and its existence in the Java language.

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 1.15

```
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 1.16

```
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 1.17

```java
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 1.18

```java
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 1.19

```java
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.

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.

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). 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. 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.

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

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

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 1.20

```java
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 1.21

```
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 1.22—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. ($\star$)
Design the `double celsiusToFahrenheit(double c)` method, which converts a temperature from Celsius to Fahrenheit.

Exercise 1.2. ($\star$)
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. ($\star$)
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. ($\star$)
Design the `double gigameterToLightsecond(double gm)` method, which converts a distance in gigameters to light seconds (i.e., distance light travels in one second). There are $1,000,000,000$ meters in a gigameter, and light travels $299,792,458$ meters per second.

Exercise 1.5. ($\star$)
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. ($\star$)
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. ($\star$)
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. ($\star$)
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. ($\star$)
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. ($\star$)
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. ($\star$)
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 1.23

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

Exercise 1.12. ($\star$)
The *z-score* is a measure of how far a given data point is away from the mean of a normally-distributed sample. In essence, roughly 68% of data falls between z-scores of $[-1, 1]$, roughly 95% falls between $[-2, 2]$, and 99.7% falls between $[-3, 3]$. This means that extreme outliers have z-scores of either less than -3 or greater than 3.

Design the `boolean isExtremeOutlier(double x, double avg, double stddev)` method that, when given a data point x, a mean μ, and a standard deviation σ, computes the corresponding z-score of x and returns whether it is an "extreme" outlier. Use the following formula:

$$Z = \frac{x - \mu}{\sigma}$$

Exercise 1.13. ($\star$)
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 1.14. ($\star$)
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 1.15. ($\star$)
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 1.16. ($\star$)
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 1.17. ($\star$)
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 1.18. ($\star$)
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 1.19. ($\star$)
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 1.20. ($\star$)
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 $z = 60$.

2. Conditionals, Recursion, and Loops

2.1 Conditionals

Decisions are otherwise called *conditionals*. We have seen conditionals before, but in this section we will reintroduce them as a concept and discuss the intricacies of Java's conditionals, including the different logical operators and behaviors thereof.

In Java, we use `if` to designate a branch in code. We supply to it a conditional expression, or a *predicate*, which resolves to either true or false. In essence, predicates resolve to boolean values. For example, if we want to return 5 if two integers a and b are the same value, we write the following:

Listing 2.1

```
static int foo() {
  int a = ...;
  int b = ...;
  if (a == b) { return 5; }
  return 0;
}
```

The == operator compares primitive values for equality, as we stated in our Java primer. If we want to use the result of a (boolean) method as the condition, we might want to inline the invocation.

Listing 2.2

```
static boolean bar() {
  ...
}

static int foo() {
  if (bar()) { return 5; }
  return 0;
}
```

We negate conditional expressions using the exclamation point operator, i.e., `!`. That is, if e is an expression that returns a boolean value, `!`e flips the output value from true to false or vice versa. We can chain conditional expressions together using the logical AND/OR operators, namely `&&` and `||` respectively. All boolean subexpressions that comprise a larger expression, conjoined by `&&`, must be true for the overall expression to be true. On the other hand, when boolean expressions are conjoined by `||`, only one must be true.

Both logical AND and logical OR are called *short-circuiting operators*. Regarding the former, if we have the expression $e = e_1$ `&&` e_2, and e_1 resolves to false, then e_2 is not evaluated, because both operands must be true for the result of the AND to be true. Logical OR works similarly; if we have the expression $e = e_1$ `||` e_2, and e_1 resolves to true, then e_2 is not evaluated, because only one operand has to be true for the result of the OR to be true.

Listing 2.3—Examples of Logical AND/OR

```java
static int foo() {
  int a = 5;
  int b = 10;
  int c = 5;
  // We never check if c == 5.
  if (a == b && c == 5) { return 100; }
  // We never check if b != 10.
  if (a == c || b != 10) { return 200; }
  return 0;
}
```

In addition to `if`, Java also has `else` and `else if` for extending the possible outcomes of a condition. When the predicate of a preceding `if` is false and an `else` block is attached, its code is evaluated. Moreover, when the predicate of a preceding `if` is false and an `else if` block is attached, the condition to the `else if` is evaluated. The former pairing represents a binary outcome, whereas the latter corresponds to more than two possible outcomes. Multiple `if` statements "stacked above one another" results in their sequential evaluation since Java assumes they are disjoint code segments. The `else if` block, on the contrary, executes only when its preceding `if` condition resolves to false. In the following code listing, we show an example of two sets of conditional statements; the former uses only `if` and the latter takes advantage of `if`, `else if`, and `else`. Accordingly, the left-hand listing returns 20 and the right-hand listing returns 10.

Listing 2.4

```java
static int foo() {
  int x = 10;
  int y = 0;
  if (x == 10) { y = 10; }
  if (y == 10) { y += 10; }
  if (x != 10 && y != 10) { y += 1000;
      }
  return y;
}
```

```java
static int foo() {
  int x = 10;
  int y = 0;
  if (x == 10) { y = 10; }
  else if (y == 10) { y += 10; }
  else { y += 1000; }
  return y;
}
```

Example 2.1. Suppose we want to translate a `String` grade into its grade-point average equivalent, treating pluses and minuses as grade increment or decrements. Our grading schema has no grade higher than a 4.0, and all failing grades result in a zero. After writing the tests, we can use a series of `if` and `else if` statements as a case analysis on the letter grade. Afterwards, once we have the base GPA according to the letter, we apply the plus or minus given the aforementioned criteria. When determining the initial GPA value, were we to use a series of `if` statements as opposed to `if/else if/else`, every predicate would need to be evaluated regardless of whether it is meaningful. By "meaningful," we mean to suggest that, for instance, once we know the GPA is a 4.0, it makes no sense to determine if the grade is a `'B'`, since we know from the previous branch that it is an `'A'`. The `else if` statements are skipped over if a preceding condition resolves to true.

Listing 2.5—GPA Calculator Testing

```java
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static GpaCalculator.gpa;

class GpaCalculatorTesting {

  @Test
  void testGpa() {
    assertAll(
      () -> assertEquals(4.0, gpa("A+")),
      () -> assertEquals(3.7, gpa("A-")),
      () -> assertEquals(0.0, gpa("F-")));
  }
}
```

Listing 2.6—GPA Calculator Testing

```java
class GpaCalculator {

  /**
   * Computes the numeric GPA for a given letter grade.
   * @param letter grade between A and F, with optional +/-.
   * @return numeric grade from 4.0 to 0.0.
   */
  static double gpa(String grade) {
    boolean plus = grade.contains("+");
    boolean minus = grade.contains("-");
    char letter = grade.charAt(0);
    double gpa = 0;

    // Compute the grade letter.
    if (letter == 'A') { gpa = 4.0; }
    else if (letter == 'B') { gpa = 3.0; }
    else if (letter == 'C') { gpa = 2.0; }
    else if (letter == 'D') { gpa = 1.0; }
    else { gpa = 0.0; }

    // Compute +/- if applicable.
    if (letter != 'A' && letter != 'F') { gpa = plus ? gpa + 0.3 : gpa; }
    if (letter != 'F') { gpa = minus ? gpa - 0.3 : gpa; }
    return gpa;
  }
}
```

The latter two `if` statements, as we said, apply increments or decrements based on whether the grade is a + or a −. We use the not-equal-to operator, `!=`, to circumvent having to apply a negation on the outside, i.e., `!(letter == 'A')`. The bodies of these cases, however, appear to be foreign, and indeed, we introduce the *ternary operator*. Because `if` is a statement, there is no way to inline a conditional into an expression. The ternary operator is a fix to this problem. We read $r = p\ ?\ c\ :\ a$ as follows: “if p is true, assign c to r, otherwise assign a to r.” Inlining conditional expressions in this fashion reduces code clutter but should be used sparingly. We could write all `if` statements as ternary operations, but doing so would obfuscate our logic.

Aside from the ‘`if`’/‘`else if`’/‘`else`’ trio, plus the ternary operator, Java has the `switch`/`case` statements, which serve to help simplify case analysis problems. A `switch` statement receives an expression e that resolves to some value v. Inside the `switch` exists `case` statements, corresponding to possible outcomes of e’s evaluation. For instance, if we wanted to write a method that determines the number of days there are in a given (non-leap year) month, we might be inclined to use several `if` statements, which is prone to errors. Let us see the answer to this problem using `switch` and `case` statements.

Listing 2.7—Month to Days Tester

```java
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static MonthToDays.monthToDays;

class MonthToDaysTester {

  @Test
  void testMonthToDays() {
    assertAll(
      () -> assertEquals(31, monthToDays("January")),
      () -> assertEquals(28, monthToDays("February")),
      () -> assertEquals(31, monthToDays("May")),
      () -> assertEquals(31, monthToDays("July")),
      () -> assertEquals(31, monthToDays("August")),
      () -> assertEquals(30, monthToDays("September")),
      () -> assertEquals(31, monthToDays("December")));
  }
}
```

Listing 2.8—Month to Days Implementation

```java
class MonthToDays {

  /**
   * Determines how many days a given month has, not accounting for leap years.
   * @param m capitalized month, e.g., "July".
   * @return the number of days in the month
   */
  static int monthToDays(String m) {
    int days = 0;
    switch (m) {
      case "February":
        days = 28;
        break;
      case "April":
      case "June":
      case "September":
      case "November":
        days = 30;
        break;
      default:
        days = 31;
```

```
    }
    return days;
  }
}
```

We evaluate m inside the `switch` statement, and it resolves to one of twelve possible strings, assuming the input is a valid and capitalized month. If, for instance, the string is `"February"`, we assign *days* to 28 and perform a `break`. Cases that comprise a `switch` block can "fall through" to the next case, meaning that if we did not insert a `break`, the program would fall all the way to the `default` case and assign 31 to *days*. Default cases correspond to "anything other data," similar to `else` blocks. In our case, we place all months that have thirty-one days in the default case to reduce the number of lines in our code. In the case of a month having thirty days, there are four possibilities, and we stack these atop one another to state that these months should have 30 assigned to *days*. If we wanted to omit the `break` statements, we might instead inline `return` statements directly, since we do not do anything aside from return *days* at the end of the method. Of course, this solution only works when the resulting target of a `switch` block *is* the desired value.

2.2 Recursion

Standard Recursion

You may or may not have seen recursion before, but in theory the concept is quite simple: a method f is *recursive* if, somewhere in the definition of f, it invokes itself. For example, in the following code segment, we define f to be a method of arbitrary arguments that calls itself from its body.

```
static int f(...) {
  ...
  f(...);
  ...
}
```

Some may question the need for recursive methods, as it appears to be circular; why would we ever want a method to call itself? There are two reasons, where the former is what we consider to be less significant than the latter:

1. It allows the programmer to repeat a given segment of code.
2. We can compose the solution to a big problem by combining the solutions to smaller problems.

So, we may certainly use recursion to repeat a task and, by transitivity, we will do that, but we primarily write recursive methods to solve some large problem by breaking it down into smaller problems that we know how to solve.

Example 2.2. Let us consider the question of addition. Consider a context where we have access to only three methods: `addOne`, `subOne`, and `isZero`, all of which are trivially defined. We also have access to conditional statements and method calls. Finally, we have an identity that $m+0=m$ for any natural number m. Here's the problem that we want to solve: we want to add two natural numbers m and n, but how do we do that? Think about how humans calculate the sum of two natural numbers (perhaps some do it differently from others, but the general process is the same). Since we do not have a + operator in this context, we have to try a different approach. Recall the identity that we have at our disposal: $m + 0 = m$. Is there a way we can make use of the identity? Imagine that we want to solve $3 + 4$ in this context. Can we rewrite this expression that takes advantage of those methods that we have at our disposal? Indeed, we can rewrite this as a series of calls to `subOne` and `addOne`, but we will first show this in math notation.

$$
\begin{aligned}
&= 3 + 4 \\
&= (3 + 3) + 1 \\
&= ((3 + 2) + 1) + 1) \\
&= (((3 + 1) + 1) + 1) + 1 \\
&= ((((3 + 0) + 1) + 1) + 1) + 1
\end{aligned}
$$

To solve $3 + 4$, we need to solve $3 + 3$, which means we need to solve $3 + 2$, which means we need to solve $3 + 1$, which means we need to solve $3 + 0$. Substituting 3 for m gives us the identity, meaning this expression resolves to m, namely 3. Recursively breaking down a problem into smaller problems is called *invoking the recursion*. Namely, we invoke the function of interest, +, inside its own definition. As part of this, we decrement n by one in attempt to head towards the identity, or the problem that we know how to solve. Such a problem is called the *base case* to our recursive method. How do we know what the base case is for this particular problem? We use our predicate for detecting if a value is zero, of course.

We still have work to do after reaching the base case, however. Even though we may substitute $3 + 0$ for 3, we have to add one to these resulting values. Let us see what this looks like.

$$
\begin{aligned}
&= ((((3 + 0) + 1) + 1) + 1) + 1 \\
&= (((3 + 1) + 1) + 1) + 1 \\
&= ((4 + 1) + 1) + 1 \\
&= (5 + 1) + 1 \\
&= 6 + 1 \\
&= 7
\end{aligned}
$$

Upon reaching the base case, using the pieces generated by the recursion, we create the solution to our overall problem. In other words, to solve $3 + 1$, we had to solve $3 + 0$, whose base case resolves to 3. We can walk back up this series of recursive calls, filling in the gaps to the previously-unknown solutions. Because $3 + 0 = 3$, we know the answer to $3 + 1$. This propagates all the way back through the recursive calls and we arrive at our desired solution of 7. Traversing through these recursive calls backwards while building the solution to the overall problem is called *unwinding the recursion*. Now that we understand the logic of our problem, we can encode it into the Java language. First, of course, we want to design our tests.

Listing 2.9—Test Class for `add` *Method*

```java
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static Add.add;
```

```
class AddTester {

  @Test
  void addTest() {
    assertAll(
      () -> assertEquals(7, add(3, 4)),
      () -> assertEquals(12, add(11, 1)),
      () -> assertEquals(6, add(0, 6)),
      () -> assertEquals(6, add(6, 0)));
  }
}
```

Listing 2.10—Implementation of `add` *Method*

```
class Add {

  static int add(int m, int n) {
    if (isZero(n)) {
      return m;
    } else {
      return addOne(add(m, subOne(n)));
    }
  }
}
```

Our recursive implementation is nothing more than restating the mathematical definition, which is certainly convenient. Let us trace through a sequence of recursive calls from a method invocation.

Is `4` zero? No! `return addOne(add(3, 3))`

Is `3` zero? No! `return addOne(add(3, 2))`

Is `2` zero? No! `return addOne(add(3, 1))`

Is `1` zero? No! `return addOne(add(3, 0))`

Is `0` zero? Yes! `return 3.`

Once we reach the base case, we unwind the recursive calls, substituting our known values for their previously-unknown values.

We now know `add(3, 0)` is 3. So, `return addOne(add(3, 0))` is `return 4`

We now know `add(3, 1)` is 4. So, `return addOne(add(3, 1))` is `return 5`

We now know `add(3, 2)` is 3. So, `return addOne(add(3, 2))` is `return 6`

We now know `add(3, 3)` is 3. So, `return addOne(add(3, 3))` is `return 7`

We now know `add(3, 4)` is 7. So, we are done.

Recursion, as we stated before, composes the solution to a large problem by first solving smaller problems.

Example 2.3. Consider the factorial mathematical operation. The factorial of a natural number n obeys the following definition:

$$0! = 1$$
$$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

What is interesting about factorial is its relation to recursion. To solve $n!$, we need to solve $(n-1)!$, which means we need to solve $(n-2)!$, all the way down to our base case of $0! = 1$. Rewriting the prior definition to instead use recursion gets us the following:

$$\begin{aligned} 0! &= 1 \\ n! &= n \cdot (n-1)! \end{aligned}$$

We should trace through a factorial invocation to see its behavior.

$$\begin{aligned} 5! &= 5 \cdot 4! \\ 4! &= 4 \cdot 3! \\ 3! &= 3 \cdot 2! \\ 2! &= 2 \cdot 1! \\ 1! &= 1 \cdot 0! \end{aligned}$$

So, after the recursive calls, we reach our base case. We still have work to do afterwards much like `add`. Rather than `addOne`, we extend our context to include multiplication for the sake of brevity, and use that as an operation. Therefore when unwinding the recursive calls we get the following trace:

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 \cdot 1 \\ 2! &= 2 \cdot 1 \\ 3! &= 3 \cdot 2 \\ 4! &= 4 \cdot 6 \\ 5! &= 5 \cdot 24 \\ &= 120 \end{aligned}$$

Now let us encode this into Java, again with tests taking precedence over the method definition.

Listing 2.11—Test Class for `fact` Method

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

class FactTester {

  @Test
  void testFact() {
    assertAll(
      () -> assertEquals(120, fact(5)),
      () -> assertEquals(1, fact(0)),
      () -> assertEquals(1, fact(1)),
      () -> assertEquals(3628800, fact(10)));
  }
}
```

Listing 2.12—Implementation of `fact` Method

```
class Factorial {

  static int fact(int n) {
    if (isZero(n)) {
      return 1;
    } else {
      return n * fact(subOne(n));
    }
  }
}
```

Once more will we derive a trace, but this time of the `fact` method.

Is 5 zero? No! `return 5 * fact(4)`
Is 4 zero? No! `return 4 * fact(3)`
Is 3 zero? No! `return 3 * fact(2)`
Is 2 zero? No! `return 2 * fact(1)`
Is 1 zero? No! `return 1 * fact(0)`
Is 0 zero? Yes! `return 1`

Upon arriving at the base case, we begin to unwind the recursive calls.

We now know `fact(0)` is 1. So, `return 1 * 1` is `return 1`
We now know `fact(1)` is 2. So, `return 2 * 1` is `return 2`
We now know `fact(2)` is 2. So, `return 3 * 2` is `return 6`
We now know `fact(3)` is 6. So, `return 4 * 6` is `return 24`
We now know `fact(4)` is 24. So, `return 5 * 24` is `return 120`
We now know `fact(5)` is 120. So, we are done.

Voilà, we get our desired solution.

Tail Recursion and Accumulators

In the previous section we discussed recursion, or what we will refer to as *standard recursion.* This style of recursion is popular because of its ease-of-use and relative correlation to mathematical definitions. Aside from this, unfortunately, there is a significant problem with standard recursion: it is a memory hog and potential recipe for disaster. The reason does not easily present itself to the programmer, and we have to dive deeper into how Java makes method calls.

Each time Java invokes a method, it pushes an *activation record* to its *method call stack.* The call stack is a location in memory where all method invocations reside. Activation records contain information about the method that was called, such as the arguments, the number of locally-defined variables, and other miscellaneous data. More importantly, activation records designate the "return location" of a method. When a method call returns, it is popped off the call stack. The stack memory, or lack thereof, is the root cause of problems with our standard recursion. Let us demonstrate this predicament with an example trace of `add` whose second argument, namely n, is incredibly large; over two million.

...
...
...
...
`add(3, 19996)`
`add(3, 19997)`
`add(3, 19998)`
`add(3, 19999)`
`add(3, 20000)`

Figure 2.1: Pushing of `add` Activation Records to the Call Stack

As we stated, calling a method pushes its activation record to the method call stack, so invoking `add(3, 2000000)` pushes one record. Then, because two million is certainly not zero, we then recursively call `add(3, 1999999)` and push that record to the call stack. This idea continues until we reach a point where there is not enough memory to push another activation record to the (call) stack, in which a `StackOverflowException` is thrown by the Java Virtual Machine. We want a way of writing recursive algorithms without having to waste so much memory and risk a stack overflow of the call stack. A potential solution to our problem is via *tail recursion* through *accumulator-passing style*.

A method f is tail-recursive if all recursive calls are in *tail position*. At first glance, this definition appears circular. But, consider this piece: an expression is in tail position if it is in the last-to-perform operation before a method return. When relating this to recursive methods, it implies that any invocation of f occurs as the last-evaluated operation prior to a return from the method. Both `add` and `fact` were non-tail recursive because each have extra work to do after the recursive calls step; that work being an unwinding of the recursive calls. Tail recursive functions do not need to unwind anything because they (for the most part) accumulate the result to an overall problem in an argument to the tail recursive method.

Example 2.4. We want to compute the factorial of some number using tail recursion. Let us design a template for this method. We know that the method must be called where the call is in tail position, so we can add this as a preliminary step. Up next we can copy the logic of the previous standard recursive algorithm with the added exception that we do not return one from the base case, but instead return an accumulated result. The goal is to construct, or generate, the factorial of some n as an argument to the method.

Listing 2.13—Template for Factorial Recursive Method

```java
class FactorialTailRecursive {

  static int factTR(int n, int acc) {
    if (isZero(n)) {
      return acc;
    } else {
      return factTR(..., ...);
    }
  }
}
```

Observe that the only change to the base case occurs in the body of the condition. So, the first argument to `factTR`, i.e., n, still trends towards the base case and, hence, should be the decrement of n. On the other hand, `acc` stores an accumulated factorial result. Consequently, we must multiply the accumulator by n, thereby with every recursive call, the accumulator approaches the correct solution.

Let us perform a trace of `factTR` to see how we build the result in the `acc` parameter. One extra factor to consider is the initial/starting value of our accumulator argument. This value depends on the context of the problem, and for factorial, the only reasonable value is one. E.g., if we initialize `acc` to zero, then we would continuously multiply and store zero as the argument to the recursive call, thereby always returning zero as the factorial of any number.

Is 5 zero? No! `return factTR(4, 5)`
Is 4 zero? No! `return factTR(3, 20)`
Is 3 zero? No! `return factTR(2, 60)`
Is 2 zero? No! `return factTR(1, 120)`
Is 1 zero? No! `return factTR(0, 120)`
Is 0 zero? Yes! `return 120`

Because we have the result, its value is simply returned from the method. We do not need to unwind the recursive calls since there is no extra work to be done after making the recursive calls in the first place. Even still, some may question how this avoids a stack overflow error because we still push an activation record to the call stack each time we invoke `factTR`, right? Indeed, this solution does not solve the stack overflow problem, because Java does not employ the necessary optimizations to do so. What might one of those solutions be, in fact? As a hypothesis, because the method is tail recursive, the Java compiler could detect this and, instead of pushing a new activation record to the call stack, it overwrites the preexisting record, hence using constant space and only one record. Overriding the existing activation record is permissible since we do not unwind the stack. Recall with standard recursion that we push an activation record to the call stack in the first place to remember the context of "how deep we are" into the recursion and what values we must substitute back into the unknowns during the unwinding phase. Conversely, when looking at the tail recursive approach, we build the result alongside heading towards the base case, meaning previous recursive calls are made irrelevant. Let's see what this looks like in the model of a stack.

The transitions between each "stack" represent the same stack wherein each represents a point in time. After the invocation of `factTR(5, 1)`, we recursively call `factTR(4, 5)` and replace the previous activation record. This follows suit until we hit the base case and return the accumulator.

One problem with tail recursion is its exposure of an accumulator to the caller of the method. The user of such a factorial function should not need to worry about what value to pass as the initial accumulator; they only want a method that computes the factorial of some natural number. The solution is to write a *driver method* and introduce *method access modifiers*. Driver methods, in short, serve to "jump start" the logic for some other, perhaps more complex, method. We should refactor the logic from `factTR` into a helper method that is inaccessible from outside the class. To do so, we affix the `private` keyword

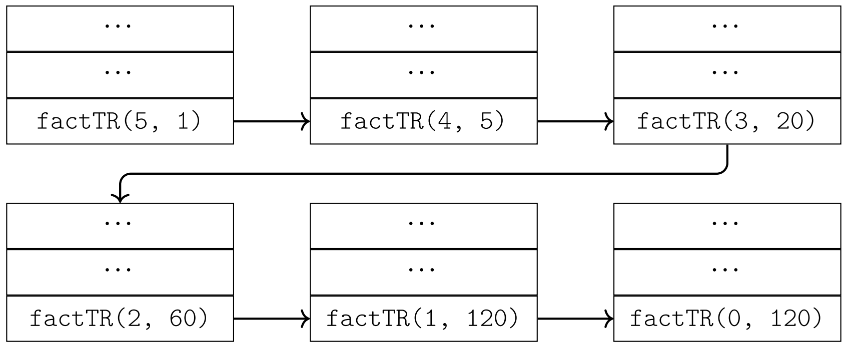


Figure 2.2: Simulated Tail Recursion with "Multiple Stacks"

in front of `static`. Private methods are unreachable/not callable from outside the class in which it is declared.

Listing 2.14

```
class FactorialTailRecursive {

  static int factTR(int n) {
    return factHelper(n, 1);
  }

  private static int factHelper(int n, int acc) {
    if (isZero(n)) {
      return acc;
    } else {
      return factHelper(subOne(n), acc * n);
    }
  }
}
```

Notice that we localized the tail recursion to this class and updated the signature of `factTR` to only have one parameter. We designate `factTR` as the driver method for jump-starting the tail recursion that occurs in `factHelper`. Driver methods, in general, should share the same signature with their standard recursion method counterparts, so as to not expose the innard implementation of a method to the caller. Hiding method implementation in this fashion is called *encapsulation*.

Example 2.5. Let us get a bit more practice using recursion by integrating strings. Suppose we want to design a method that removes all characters whose position is a multiple of three. For example, given the string `"ABCDEFGHI"`, we want to return `"ABDEGH"`, since `"C"`, `"F"`, and `"I"` are located at positions (note the use of position and not index) are divisible by three. Tests are, of course, warranted and necessary.

Listing 2.15—Remove Divisible By Three Tests

```
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static DivByThree.removeDiv3Chars;

class DivByThreeTests {
```

```java
  @Test
  void removeDiv3CharsTest() {
    assertAll(
      () -> assertEquals("ABDEGH", removeDiv3Chars("ABCDEFGHI")),
      () -> assertEquals("CCC", removeDiv3Chars("CC")),
      () -> assertEquals("AB", removeDiv3Chars("AB")),
      () -> assertEquals("A", removeDiv3Chars("A")),
      () -> assertEquals("", removeDiv3Chars("")),
      () -> assertEquals("ABCD", removeDiv3Chars("ABD")));
  }
}
```

We can break our input down into two cases: when the string does not have at least three characters, and otherwise. If the string has less than three characters, we return the string itself. Otherwise, we want to compose a new string containing the first two characters, skipping the third, and recursing on the rest. In the "otherwise" case, we are guaranteed that the input string has at least three characters, implying that `substring(3)` will not fail. Because the `substring` method of one argument is exclusive, if the provided index is the end of the string, the empty string is returned.

Listing 2.16—Remove Divisible by Three Implementation

```java
class DivByThree {

  static String removeDiv3Chars(String s) {
    if (s.length() < 3) {
      return s;
    } else {
      return s.substring(0, 2) + removeDiv3Chars(s.substring(3));
    }
  }
}
```

Thinking recursively takes time, and there is no better way to get better than extensive practice. Let us now convert the method into its tail recursive counterpart. Due to the trivial nature of writing tests, we will omit them for our tail recursive version. The algorithm is largely the same, except for the added accumulator that builds the resulting string instead of relying on the recursive unwinding to occur. Our base case concatenates s onto the end of the accumulator.

Listing 2.17—Tail Recursive Remove Divisible by Three Implementation

```java
class DivByThree {

  static String removeDiv3CharsTR(String s) {
    return removeDiv3CharsHelper(s, "");
  }

  private static String removeDiv3CharsHelper(String s, String acc) {
    if (s.length() < 3) {
      return acc + s;
    } else {
      return removeDiv3CharsHelper(s.substring(3), acc + s.substring(0, 2));
    }
  }
}
```

So, we have explored both standard and tail recursive methods, and how a programming language might optimize tail recursive calls. The thing is, tail recursion has a direct correspondence to loops, i.e., `while`. In fact, some programming languages convert all tail recursive functions into their iterative counterparts, alleviating the need for a stack whatsoever. Replacing tail recursion, or tail recursive calls, with iteration is known as *tail call optimization*.[1] In the next section, we will discuss a translation pipeline from tail recursion to loops in greater detail, as well as describe the syntax and semantics of Java iteration structures.

2.3 Loops

Looping is a fundamental concept in computer programming. Loops allow for repetition of actions or tasks. As we stated in the previous section on recursion, any tail recursive algorithm may be translated into an algorithm that uses loops. In this section, we will describe this translation pipeline as a sequence of steps, then begin to distance ourselves from recursion when it is suboptimal.

Translation Pipeline for Tail Recursive Methods

A Coarse-Grained Approach

What follows is a high-level introduction to converting from tail recursive methods to iteration. While you may not understand everything at first, we supplement this with a comprehensive translation schema.

Writing recursive methods is certainly fun.[2] Though, the use of recursion is not always the most intuitive approach to solve a problem according to some programmers/students. Many programming languages offer *iteration* statements, which allow us to perform a task that we might otherwise use recursion to complete. Suppose we have a standard recursive *fact* method.

Listing 2.18—Standard Recursive Factorial

```
static int fact(int n) {
  if (isZero(n)) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

```
static int factTR(int n, int acc) {
  if (isZero(n)) {
    return acc;
  } else {
    return factTR(subOne(n), n * acc);
  }
}
```

The first step in converting a recursive method into its iterative counterpart is to rewrite it using tail recursion. Something we, of course, already know how to do from the previous section. Notice that, in order to “tail-recursify” `fact`, we had to add an extra parameter that keeps track of the “current result.”[3] The reason for converting recursive methods into

[1] Java is one of many imperative languages that does not support tail call optimization, meaning that tail calls, unfortunately, continue to blow up the procedure call stack.

[2] The definition of “fun” is, of course, relative.

[3] Some tail recursive methods need more than one extra parameter—it is a case-by-case basis.

tail recursion is because of their direct relation to iteration.[1] Let us look a little deeper into this and find out why.

Our iterative version of `fact` should move all "accumulator" variables into the method body. E.g., `acc` will now be declared locally to the `fact` method. In addition to this movement, all accumulator-to-local variables should have an "iterative purpose statement," which mimics the documentation comment for the accumulator parameter.

Listing 2.19—Localizing Accumulator Variables

```
static int fact(int n) {
  // acc stores the current factorial value as n goes to zero.
  int acc = 1;
  // TODO.
}
```

Second, we must describe the syntax of a loop. A `while` loop is the construct of choice, and it has two components: a condition denoted as a predicate, and a body. The loop checks to see if the given predicate is true and, if so, executes the body of the loop. On the other hand, if the predicate is false, we jump down to below the loop. Each pass through the loop, it re-verifies that the predicate holds true. Unlike expressions, however, a `while` loop, itself, does not resolve to some value. It is called a *statement* because of this property. Let us begin by defining the predicate of our loop. To answer this, we ask, then answer, the question of the base case(s) for our tail recursive method. As we see, our base case is true when n is zero. Therefore our loop should continue to execute as long as n is not zero. One tail recursive method call correlates directly with one loop iteration. So, let us remove the `if/else` statement chain and substitute them by a loop whose condition is nothing more than the negated base case(s).

Listing 2.20—Adding the Loop and Negated Predicate

```
static int fact(n) {
  // acc stores the current factorial value as n goes to zero.
  int acc = 1;
  while (!(n == 0)) {
    // TODO.
  }
}
```

Finally, we come to the heart of the loop. Within, we update variables according to how they are updated in the tail recursive call. Namely, as we saw, n is decremented by 1, and *acc* is multiplied by n. We must be careful, however, because the order of these statements is significant! In the tail recursive method call to `fact`, the n that is decremented is passed to the method, whereas the original value of n is used when multiplying by *acc*. As a result, the accumulator update should come first.

Listing 2.21—Loop Body Becomes Tail Recursive Updates

```
static int fact(int n) {
  // acc stores the current factorial value as n goes to zero.
  int acc = 1;
  while (!(n == 0)) {
    acc = acc * n;
```

[1] A method definition may already be tail recursive depending on the circumstance.

```
    n = n - 1;
  }
}
```

We are almost done. The loop body is now complete, with each modification pairing precisely with some piece of the tail recursive version. All that remains is the base case return statement. In our tail recursive method, once we hit the base case, we return `acc`. We model this, of course, using a return outside the loop.

Listing 2.22—Return the Base Case

```
static int fact(int n) {
  // acc stores the current factorial value as n goes to zero.
  int acc = 1;
  while (!(n == 0)) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}
```

Excellent, we now have an iterative version of the factorial method! Let us compare these two side-by-side, color-coding their similarities. Base cases are red, accumulated variables/steps are yellow, and return values are green.

Listing 2.23—Tail Recursive versus Iterative Factorial

```
static int fact(int n, int acc) {
  if (n == 0) {
    return acc;
  } else {
    return fact(n - 1, acc * n);
  }
}
```

```
static int fact(int n) {
  // acc stores the current factorial
  // value as n goes to zero.
  int acc = 1;
  while (!(n == 0)) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}
```

A Fine-Grained Approach

What we just saw was a fast-paced, high-level overview of the conversion process from tail recursive methods into methods that use while loops. Let's take a step back and slow our approach to better understand each piece. We first want to describe a general outline of the steps to success in this translation pipeline.

Of course, the goal, in due time, is to work our way from TR to I, but there are a few highly important intermediary components to this process. Note that the lines from P to R, and R to TR are not as important for this section of the transformation.

TR is the tail recursive method derived either from the problem statement P or the standard recursive step R. From here, we make our way to $TR_{\mathbf{H}}$, denoting the "color-coding" phase, where we mark the three sub components of a tail recursive method, those being the base case(s), updated variables in the tail recursive call, and returned values from the base case.

$TR_{\mathbf{IVP}}$, or "Iterative Variable Purpose," is a step following the method signature, but preceding the loop definition. In this step, we examine the updated variables/accumulators

$$P \xrightarrow{1} R \xrightarrow{2} TR \xrightarrow{3} TR_{\mathbf{H}} \xrightarrow{4} TR_{\mathbf{IVP}} \xrightarrow{5} \left(TR_{\mathbf{C}} \rightarrow TR_{\mathbf{B}} \rightarrow TR_{\mathbf{Ret}} \right) \dashrightarrow I$$

Figure 2.3: Fine-Grained Translation Pipeline

marked in $TR_{\mathbf{H}}$ and localize them into variables not passed as arguments to a method, but rather as a sequence of value updates. Moreover, we add comments to these variable declarations explaining their purpose.

$TR_{\mathbf{C}}$ is the step wherein we write the `while` keyword, followed by the negated base case condition(s) as a series of conjunctions.

$TR_{\mathbf{B}}$ is where we design the body of our loop, which contains update statements to our localized iterative variables rather than arguments to a recursive call.

Finally, in $TR_{\mathbf{Ret}}$, we add the line to return the accumulated local variable(s). I is the output translation.

$TR_{\mathbf{H}}$:

When designing a tail recursive method, there are several values to keep in mind: base cases (i.e., terminating conditions), returned values that are not method calls, and accumulators. Each of these play a crucial role in the transition to I, and quickly yet correctly identifying these as they fit in tail recursive methods is paramount. Let us re-look at our old factorial friend to see what this entails.

Listing 2.24—Tail Recursive Factorial

```
static int fact(int n, int acc) {
  if (n == 0) {
    return acc;
  } else {
    return fact(n - 1, acc * n);
  }
}
```

Marking the base case(s) is usually rather simple, as they are most often the selection statements that return a value rather than a recursive method invocation. So, the only instance of this in the above definition is $n == 0$. So, we highlight this in a color, e.g., red.

Listing 2.25—Highlighting Factorial Base Case

```
static int fact(int n, int acc) {
  if (n == 0) {
    return acc;
  } else {
    return fact(n - 1, acc * n);
  }
}
```

To coincide with the base case(s), we also want to highlight the returned values that are not recursive calls. Only one exists, namely *acc*. Let us highlight this in green.

Listing 2.26—Highlighting Factorial Returned Values

```
static int fact(int n, int acc) {
  if (n == 0) {
    return acc;
  } else {
    return fact(n - 1, acc * n);
  }
}
```

We are almost done. Now, we want to highlight variables passed to the (tail) recursive call that are updated. When we say updated, we mean "modified" insofar as they are not simply copied verbatim, e.g., $f(n) = f(n)$, in which we see n remains unaltered. Fortunately, both n and *acc* are updated (n is decremented by one; *acc* is multiplied by n). Let us highlight these changes in yellow.

Listing 2.27—Highlighting Factorial Updated Arguments

```
static int fact(int n, int acc) {
  if (n == 0) {
    return acc;
  } else {
    return fact(n - 1, acc * n);
  }
}
```

$TR_{\mathbf{IVP}}$:

All accumulator variables in tail recursive methods serve some purpose, one way or another, hence their necessity. The same holds true for local variables defined to substitute accumulators. Conveniently enough, every variable designated as an accumulator morphs nicely into a local variable when writing the iterative counterpart. Consider, once more, the tail recursive factorial definition. We use *acc* to accumulate the result as a parameter, indicating the need for an accumulator statement. We can simply insert this as an addendum to our Java documentation comment.

Listing 2.28—Accumulator Statement for Factorial

```
/**
 * @accumulator acc stores the current factorial product.
 */
static int fact(int n, int acc) {
  if (n == 0) {
    return acc;
  } else {
    return fact(n - 1, acc * n);
  }
}
```

In the translation to a loop, we move *acc* to the body of the method, and write a similar iterative variable purpose.[1] Note that the value the initialized variable receives depends

[1] By "move," we mean "remove" but not "delete."

on the problem/context, but it always matches whatever value is passed to a tail recursive helper method.

Listing 2.29—Iterative Variable Purpose Statement

```
static int fact(int n) {
  // acc stores the current factorial product.
  int acc = 1;
}
```

Writing these imperative variable purposes, akin to accumulator statements, helps us organize what variables change and, more importantly, when and how they change.

TR_C:

Up next is where we take our base case condition, highlighted in red, and insert it as the negated condition for our loop. First, we add the `while` keyword to our method, then follow this with a set of parentheses and an exclamation point immediately after the opening parenthesis but before the closing. Inside these parentheses we insert the base case condition. To be safe, you should also insert parentheses for the base case, which ensures that the correct expression is negated by the logical 'not' operator. Follow this with an opening and closing brace set.

Listing 2.30—Adding Condition to Loop

```
static int fact(int n) {
  // acc stores the current factorial product.
  int acc = 1;
  while (!(n == 0)) {
    // TODO.
  }
}
```

TR_B:

Finally we get to the fun part of this translation process: the body of our loop. Here we need to make a design choice of what variables to update and when they should be updated. We take the yellow highlighted tail recursive arguments, create assignment statements out of them, and insert them into the body of our loop. To do so, we need to follow two principles:

Rule of Reassignment: In any tail recursive call, if we pass an expression e which updates parameter p, then in the loop body, we directly reassign p to e.

Rule of Update: If we have two parameters p and q that are updated as part of the tail recursive call, and p's value is used as part of updating q, then q must be modified before p.

The rule of reassignment is straightforward: we have an expression that resolves to some value, which corresponds precisely to those highlighted arguments. This expression is converted into an assignment statement to the locally-declared variable.

The rule of update, on the other hand, is not as straightforward. Essentially, we use this rule to ensure that variables whose value depends on another are not prematurely updated. Consider the following incorrect update of n before updating *acc*:

```
n = n - 1
acc = acc * n
```

Iteration #	n	acc
0	5	1
1	4	4
2	3	12
3	2	24
4	1	24
5	0	0

We see that this variable update ordering produces 0, which does not match our recursive trace! We get zero thanks in part due to the final multiplication before our loop condition is falsified. Let us now try the other possible ordering, in which *acc* is updated before n. Hence, we are now in the second attempt of completing TR_B:

```
acc = acc * n
n = n - 1
```

Iteration #	n	acc
0	5	1
1	4	5
2	3	20
3	2	60
4	1	120
5	0	120

This results in 120, which matches our recursive trace. Therefore, without loss of generality, we can conclude that we should update the accumulator variable before updating *n* in this circumstance.

Determining the correct order of update, according to the rules we specify, may take a few tries to get right. The idea is to match the result of a tail recursive trace done previously that is known to be correct.

$TR_{\textbf{Ret}}$:

In our final stage of translation, we add the necessary return statement(s) that serve to return the accumulated result from our loop. We highlighted these values in green during the highlighting/color-coding stage.

Listing 2.31—Adding Accumulator Return Statement to Loop

```java
static int fact(int n) {
  // acc stores the current factorial product.
  int acc = 1;
  while (!(n == 0)) {
    acc = acc * n;
    n = n - 1;
  }
  return acc;
}
```

And that is it; the translation pipeline is complete, and we now know how to translate a tail recursive method into one that uses a loop.

The astute reader might question the need for a tail recursion-to-iteration translation schema. We mentioned the term tail call optimization in the previous section on recursion, and will now explain the relation to loops.

Recall the benefit of tail recursion over standard recursion: it uses only one (replaceable) activation record. Though, because we can translate any tail recursive method into an iterative algorithm, we forgo the stack in its entirety.

From here we might ask a similar question: can we translate any standard recursive method into one that uses tail recursion (and by transitivity, iteration)? In general, the answer is yes, through a concept called *continuation-passing style*. Because of how difficult it is to implement continuations in Java, we will omit any further discussion, but interested readers should delve into functional programming if this equivalence is intriguing.

Iteration Constructs

Perhaps the related equivalence to tail recursion is too abstract for some to digest. Indeed, we believe the relationship is somewhat far-fetched at first glance, but after enough practice, it becomes clear. We will now discuss loops from a non-translation perspective. That is, if we assume the translation diagram from before, we are going straight from the problem statement P to the iterative solution I.

As we stated, loops are statements in Java, meaning they do not, themselves, resolve to a value. Therefore any and all lines of code executed inside the body of a loop must be statements rather than expressions.

Example 2.6. Suppose we want to print the first one hundred prime integers. Recall the definition of primality: a positive integer n is prime if it is divisible by only one and itself. Without loops, we would otherwise solve this task using recursion. Although summing prime values is not a too terribly complicated task, it requires a bit of a verbose set of methods, should we choose to do it tail recursively. Instead, let us try and use loops to our advantage. We want to continue looping until we have summed one hundred prime values. Fortunately there is exactly one correct test case for this method, so writing an elaborate test suite is superfluous.

Declaring an integer counter c is mandatory. Our loop condition, therefore, is to continue so long as we have not reached one hundred prime values. When designing loop conditions, we ask ourselves the question, "What should be true after the loop?," and design the condition around the answer. Our counter c will be exactly one hundred upon loop completion, so the condition is the negated version of this expression.[1]

Up next we need a method to determine if a value is prime. We presented the definition of primality before, and it is also known that we only need to check up to the square root of n for primality. Let us design such a method, writing the appropriate tests. Our condition is to continue so long as a variable, i, is less than or equal to the square root of our input number. The variable i serves as a counter towards our number; if we find that n is not divisible by any numbers from two up to the square root of n, then it must be prime.[2] Note the edge cases of zero and one, which are both non-prime.

---

[1]Rather than `(!(c == 100))`, we can write `(c != 100)` to achieve the same effect.

[2]We cast the result of `Math.sqrt` to an integer because it returns a `double` rather than an `int` datatype.

Listing 2.32—Sum Of First 100 Primes Skeleton

```
class SumPrimes {

  /**
   * Computes the sum of the first 100 prime integers.
   * @return sum of first 100 primes.
   */
  static int sum100Primes() {
    int c = 0;
    while (c != 100) { /* TODO. */ }
    return 0;
  }
}
```

Listing 2.33—Loop Primality Tests

```
import static Assertions.assertAll;
import static Assertions.assertTrue;
import static Assertions.assertFalse;
import static Prime.isPrime;

class PrimeTester {

  @Test
  void testPrime() {
    assertAll(
      () -> assertTrue(isPrime(17)),
      () -> assertTrue(isPrime(101)),
      () -> assertTrue(isPrime(3)),
      () -> assertTrue(isPrime(9173)),
      () -> assertFalse(isPrime(0)),
      () -> assertFalse(isPrime(1)),
      () -> assertFalse(isPrime(2)),
      () -> assertFalse(isPrime(202)),
      () -> assertFalse(isPrime(213123447)));
  }
}
```

Listing 2.34—Loop Primality Implementation

```
class Prime {

  /**
   * Determines if a given integer is prime.
   *
   * @param n positive integer.
   * @return true if prime, false otherwise.
   */
  static boolean isPrime(int n) {
    if (n < 2) {
      return false;
    } else {
      int bound = (int) Math.sqrt(n);
      int i = 2;
      while (i <= bound) {
        if (n % i == 0) {
          return false;
        }
        i++;
```

```
      }
      return true;
    }
  }
}
```

The primality tests pass, which means we can put `isPrime` to work inside of `sum100Primes`. The method logic is straightforward: initialize our counter c at one. We also need a variable to track the last-found prime integer, so we know from where to start the search. We will call this variable l and initialize it to zero. Finally, a *sum* variable accumulates the sum of each prime. Inside the loop, we declare another variable v, which serves as the value to check for primality, taking on the value of $l + 1$ (if it were just l, we would forever check the same value for primality!). Run v through the `isPrime` method and, if it is prime, assign to l the value of v, add v to the sum, and increment c. Otherwise, increment l by one.

Listing 2.35—Sum Of First 100 Primes Implementation

```
class SumPrimes {

  /**
   * Computes the sum of the first 100 prime integers.
   * @return sum of first 100 primes.
   */
  static int sum100Primes() {
    // Counter for no.  of primes.
    int c = 0;
    // Current "prime value".
    int l = 0;
    // Running sum.
    int sum = 0;

    while (c != 100) {
      int v = l + 1;
      if (isPrime(v)) {
        l = v;
        sum += v;
        c++;
      } else {
        l++;
      }
    }
    return sum;
  }
}
```

Running the code produces 24133: the desired answer.

Example 2.7. Suppose we want to compute the sum of the first n odd reciprocals. That is, $\frac{1}{1} + \frac{1}{3} + \frac{1}{5} + \cdots + \frac{1}{2n-1}$. Again, we might do this recursively, but let us try and write an iterative algorithm. The signature is straightforward: we want to receive some integer n and compute the sum of n odd reciprocals. Our tests are trivial to write with the employment of a calculator.

Listing 2.36—Odd Reciprocal Summation Tests

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

```
import static OddReciprocalSum.oddRecipSum;

class OddReciprocalSumTester {

  private static final double DELTA = 0.001;

  @Test
  void testOddRecipSum() {
    assertAll(
      // 1/1 + 1/3 + 1/5 + 1/7.
      () -> assertEquals(1.67619, oddRecipSum(4), DELTA),
      // Sum of zero reciprocals is zero.
      () -> assertEquals(0, oddRecipSum(0), DELTA),
      // 1/1.
      () -> assertEquals(1, oddRecipSum(1), DELTA),
      // 1/1 + 1/3 + 1/5 + 1/7 + 1/9 + 1/13 + ...
      () -> assertEquals(2.26435, oddRecipSum(13), DELTA));
  }
}
```

The implementation is similarly simple: we create a counter c that increments so long as c is not equal to n, accumulating the *sum* of 1 divided by $2c - 1$.

Listing 2.37—Odd Reciprocal Summation Implementation

```
class OddReciprocalSum {

  /**
   * Computes the sum of the first n odd reciprocals.
   * @param n number of reciprocals to compute.
   * @param sum of n odd reciprocals.
   * @return sum of first n odd reciprocals.
   */
  static double oddRecipSum(int n) {
    int c = 0;
    double sum = 0;
    while (c != n) {
      sum += c / (double) (2 * c - 1)
      c++;
    }
    return sum;
  }
}
```

Note the cast of the denominator from an `int` to a `double`; what happens if we omit the cast? Our tests fail because dividing two integers produces another integer, which is not desired when all of our numbers are less than or equal to one.[1]

While loops are reserved for when cases of termination are unknown. That is, we may or may not know when a while loop condition becomes false. Thus far, all methods that we have written use deterministic and predictable termination cases; we increment a counter until hitting some upper-bound. The use of `while` is unnecessary in these circumstances due to the equivalent `for` construct.

With a `for` loop, we provide three values: an *initializer statement*, a conditional expression, and a *step statement*. We have previously seen conditional expressions as they are

[1]Equivalently, we might multiply c by 2.0, or subtract 1.0; all of these coerce the `int` into a `double`, thereby allowing correct division.

the basis for our `while` condition. An initializer statement, as its name suggests, declares a variable and initializes it to some value. Stepping statements determine how to change a value between iterations of the loop. In our `while` loops, notice how we always incremented the counter by one at the end of the loop. Stepping statements take care of this for us, reducing the need for such simple statements in the loop body.

Example 2.8. Suppose we wish to write a program that returns the sum of the integers between two integers a and b, inclusive. We know how many integers there are between a and b, assuming $a \leq b$, so we should use a `for` loop to solve this problem.

Listing 2.38—Sum of Integers in the Interval $[a, b]$

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

class SumIntervalTester {

  @Test
  void testSumInterval() {
    assertAll(
      () -> assertEquals(0, sum(5, 0)),
      () -> assertEquals(0, sum(0, 0)),
      () -> assertEquals(55, sum(0, 10)),
      () -> assertEquals(55, sum(1, 10)),
      () -> assertEquals(110, sum(5, 15)));
  }
}
```

We want to iterate from $i = a$ so long as $i \leq b$. We use the variable i out of convention; one could easily use any other unused identifier.

Listing 2.39—Sum of Integers in the Interval $[a, b]$

```
class SumInterval {

  /**
   * Computes the sum of the integers between a and b inclusive.
   * @param a lower-bound inclusive.
   * @param b upper-bound inclusive.
   * @return sum of values.
   */
  static int sum(int a, int b) {
    if (a > b) { return 0; }
    else {
      int sum = 0;
      for (int i = a; i <= b; i++) { sum += i; }
      return sum;
    }
  }
}
```

Example 2.9. Let's write an example of an unpredictable loop; one that best suits the use of `while`. Suppose we want to manually compute the square root of some positive value, i.e., without Java's implementation. A simple algorithm to use is *Newton's Approximation*:

$$g_{x+1} = \frac{\left(g_x + \dfrac{n}{g_x}\right)}{2}$$

Where g_x is called a “guess.” The idea is to continuously apply this formula until we are “close enough” to the square root. Close-enough is a heuristic whose value, when very small, increases the running time of our loop. Small values of g_x indicate a closer approximation to the true square root of n. For instance, suppose $n = 64$ and $\Delta = 0.1$. Applying the formula (initializing g_0 to n) gets us the following trace:

$$
\begin{aligned}
g_1 &= \frac{\left(64 + \dfrac{64}{64}\right)}{2} && = 32.500 \\
g_2 &= \frac{\left(32.5 + \dfrac{64}{32.5}\right)}{2} && = 17.234 \\
g_3 &= \frac{\left(17.234 + \dfrac{64}{17.234}\right)}{2} && = 10.474 \\
g_4 &= \frac{\left(10.474 + \dfrac{64}{10.474}\right)}{2} && = 8.2921 \\
g_5 &= \frac{\left(8.2921 + \dfrac{64}{8.2921}\right)}{2} && = 8.0051
\end{aligned}
$$

At each iteration, we square the current guess and determine if the absolute difference between it and n is less than the guess. When so, that iteration of g_x is the square root approximation. Otherwise, we continue to approach the square root value. We cannot reasonably predict when this will occur (that is, how many iterations are necessary), so we use a `while` loop. Because there are multiple potential points of failure in this algorithm, we make sure to write an extensive test suite. The method itself is a re-telling of the mathematical definition with bits of Java syntax sprinkled about.

Listing 2.40—Newton’s Approximation Tests

```java
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static NewtonApproximation.sqrtApprox;

class NewtonApproximationTester {

  final double DELTA = 0.1;

  @Test
  void testNewtonApproximation() {
    assertAll(
      () -> assertEquals(Math.sqrt(0), sqrtApprox(0, DELTA)),
```

```
        () -> assertEquals(Math.sqrt(1), sqrtApprox(1, DELTA)),
        () -> assertEquals(Math.sqrt(3), sqrtApprox(3, DELTA)),
        () -> assertEquals(Math.sqrt(16), sqrtApprox(16, DELTA)),
        () -> assertEquals(Math.sqrt(64), sqrtApprox(64, DELTA)),
        () -> assertEquals(Math.sqrt(256), sqrtApprox(256, DELTA)),
        () -> assertEquals(Math.sqrt(4000), sqrtApprox(4000, DELTA)),
        () -> assertEquals(Math.sqrt(129500), sqrtApprox(1295000, DELTA)));
  }
}
```

Listing 2.41—Newton's Approximation Algorithm

```java
class NewtonApproximation {

  /**
   * Computes an approximation of the square root of
   * a number using Newton's Approximation algorithm.
   * @param n number to square root.
   * @param delta approximation range.
   * @return approximation of sqrt of n.
   */
  static double sqrtApprox(double n, double delta) {
    double g = n;
    // As long as the guess is too far from the real value...
    while (Math.abs(g * g - n) > delta) {
      g = (g + (n / g)) / 2;
    }
    return g;
  }
}
```

To decrease the level of accuracy, we of course should pass a different value for the *delta* argument. Conversely, to see a higher level of precision, a *delta* that is closer (but not equal to) to zero is necessary. Our implementation of the square root algorithm is far from optimal, but it exists to demonstrate the necessity of `while` loops. Though it is important to note that the expressive power of both `for` and `while` is largely irrelevant if we only concern ourselves with their semantics; they are equal in power. This means that any `for` loop is representable with a `while` and vice-versa. Depending on the circumstance, however, one might be preferable over the other. We reiterate that `for` loops should be used when we know how many iterations there are for a certain task, whereas `while` loops are for indeterminate termination conditions.

Example 2.10. We will write one more example of first writing a recursive algorithm, then its tail recursive version, then its iterative counterpart. Suppose we want to write a method to determine the n^{th} Fibonacci number. The *Fibonacci sequence* is defined recursively as follows:

$$
\begin{aligned}
Fib(0) &= 0 \\
Fib(1) &= 1 \\
Fib(n) &= Fib(n-1) + Fib(n-2)
\end{aligned}
$$

Writing tests for a Fibonacci algorithm is trivial to do and propagates through to the other variants. So, writing tests for the standard recursive algorithm, assuming they are correct, allows us to easily write tests for the others!

Listing 2.42—Standard Recursion Fibonacci Algorithm Tests

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

class FibonacciTester {

  @Test
  void testFib() {
    assertAll(
      () -> assertEquals(0, fib(0)),
      () -> assertEquals(1, fib(1)),
      () -> assertEquals(1, fib(2)),
      () -> assertEquals(8, fib(5)),
      () -> assertEquals(55, fib(10)),
      () -> assertEquals(6765, fib(20)));
  }
}
```

The standard recursive algorithm is trivial to write but also horribly inefficient; because we make two recursive calls to *Fib*, we end up computing the same data multiple times, resulting in an *exponential time algorithm*. In other words, small inputs, e.g., *Fib*(40), will take a very long time to finish.

Listing 2.43—Standard Recursive Fibonacci Implementation

```
class Fibonacci {

  static int fib(int n) {
    if (n <= 1) { return n; }
    else { return fib(n - 1) + fib(n - 2); }
  }
}
```

Our method works and tests pass, which is great, but we can do better by using tail recursion. The tail recursive variant, however, is slightly more complicated than prior tail recursive definitions due to the use of two accumulators. We use two values to compute the "next" Fibonacci number: the previous, and the previous' previous. So, storing these as values, say, a and b, is sensible, where a is the "previous' previous," and b is the previous. Thus, when making a tail recursive call, we assign b as $a + b$, and a as b. We must not forget the driver method!

Listing 2.44—Tail Recursive Fibonacci Implementation

```
class Fibonacci {

  static int fibTR(int n) {
    return fibHelper(n, 0, 1);
  }

  /**
   * Computes the nth Fibonacci number using tail recursion.
   * @param n nth Fibonacci number.
   * @accumulator a "previous' previous" fib.
   * @accumulator b "previous" fib.
   * @return result.
   */
```

```java
  static int fibHelper(int n, int a, int b) {
    if (n == 0) {
      return a;
    } else if (n == 1) {
      return b;
    } else {
      return fibHelper(n - 1, b, a + b);
    }
  }
}
```

Our tail recursive implementation has two base cases instead of one due to the two base cases of the mathematical recursive definition. It may be a little difficult to visualize the initial accumulator values (i.e., the starting values of a and b), but just consider the base case(s) of the standard recursive algorithm and let this/these guide your decision.

Finally we arrive at the iterative algorithm. To avoid any confusion in the conversion process, we will use our translation pipeline. Skipping a few steps up to TR_C:

Listing 2.45—Initial Translation of Fibonacci TR To I

```java
static int fibHelper(int n,
                     int a,
                     int b) {
  if (n == 0) {
    return a;
  } else if (n == 1) {
    return b;
  } else {
    return fibHelper(n - 1, b, a + b);
  }
}
```

```java
static int fib(int n) {
  // Holds the "previous' previous".
  int a = 0;
  // Holds the "previous".
  int b = 1;
  while (???) {
    // TODO.
  }
}
```

The question now is, what do we use as the `while` loop condition? We are obviously decreasing n by one with each (tail) recursive call, but since we have two base cases, what do we do? Realistically, we only need to use the loop for one base case: namely $n == 1$, since the only other possibility for n is if it is zero, in which we return zero. Therefore, our loop conditional is the negated condition of $n == 1$, namely $(!(n == 1))$. From here, we run into another problem: according to the rule of update, since b's value depends on a, we need to update b before a, but because a's value depends on b, we are at a bit of an impasse. Fear not, because we have an amendment to TR_B, which adds the *rule of temporary*.

Rule of Temporary: If two parameters p and q depend on each other, introduce a temporary variable t_p, assign p to t_p, update p, then update q using t_p in place of p.

In essence, we generate a local temporary variable to hold onto the old value of one of our accumulators. Following this logic, we create the variable t_a, assign to it a, update a, then update b using t_a in place of a. Since n does not a dependent variable, its ordering in the mix is irrelevant.

The last stage is simple; we return the result of the other base case, namely b when n is finally one.

Listing 2.46—Iterative Fibonacci Implementation

```java
static int fib(int n) {
  // Holds the "previous' previous".
```

```java
    int a = 0;
    // Holds the "previous".
    int b = 1;
    // Initial base case.
    if (n == 0) {
      return a;
    } else {
      while (!(n == 1)) {
        int ta = a;
        a = b;
        b = ta + b;
        n = n - 1;
      }
      return b;
    }
  }
```

The Fibonacci sequence is predictable in that we know how many iterations are necessary to compute a result. Therefore, a `for` loop seems like a reasonable substitute to `while` in this circumstance.

We still must declare variables a and b, but instead of decrementing n, we can instead localize an accumulator i (initialized as n) to, counter-intuitively, decrement towards the base case.

No significant changes in the implementation are made, aside from a new local variable to count down from n until it hits one.

Listing 2.47—Iterative Fibonacci Implementation Using `for`

```java
  static int fib(int n) {
    int a = 0;
    int b = 1;
    if (a == 0) {
      return a;
    } else {
      for (int i = n; !(n == 1); i--) {
        int ta = a;
        a = b;
        b = ta + b;
      }
    }
    return b;
  }
```

Example 2.11. We can use two keywords to alter the control flow of a loop: `break` and `continue`. We saw `break` when working with the `switch` keyword and statement, but in the context of loops its meaning has more significance. Suppose that we're writing the `int findFirstVowel(String s)` method that returns the first occurrence of a vowel in a string s, and -1 if it has no vowels. The `break` statement will be used to exit the loop body early upon finding a vowel. We will traverse over the string characters using a `for` loop and terminate if we reach the end of the string, hence the inclusion of the `i < s.length()` condition. Our implementation must also include a method to determine whether or not a character is a vowel. We could write ten separate clauses for checking if a character is the uppercased or lowercased version of the vowel, or we could eliminate half by simply converting the character to either case and only checking those.

Listing 2.48

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

class FindFirstVowelTester {

  @Test
  void testFindFirstVowel() {
    assertAll(
      () -> assertEquals(-1, findFirstVowel("")),
      () -> assertEquals(-1, findFirstVowel("BDFG")),
      () -> assertEquals(-1, findFirstVowel("111122223333")),
      () -> assertEquals(0, findFirstVowel("ABCD")),
      () -> assertEquals(0, findFirstVowel("aBCd")),
      () -> assertEquals(1, findFirstVowel("bEdFGhiOnP")),
      () -> assertEquals(7, findFirstVowel("BDFGHJKO")),
    )
  }
}
```

Listing 2.49

```
class FindFirstVowel {

  /**
   * Finds the index of the first vowel in a given string. If there are no vowels
   * in the string, we return -1.
   * @param s - input string.
   * @return index of first vowel occurrence, or -1 if it has no vowels.
   */
  static int findFirstVowel(String s) {
    int idx = -1;
    for (int i = 0; i < s.length(); i++) {
      if (isVowel(s.charAt(i))) {
        idx = i;
        break;
      }
    }
    return idx;
  }

  /**
   * Determines whether or not the character is a vowel. We convert it to
       lowercase
   * then check it against those five vowels.
   * @param ch - character to check.
   * @return true if ch is a vowel, false otherwise.
   */
  static boolean isVowel(char ch) {
    char lch = Character.toLowercase(ch);
    return ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u';
  }
}
```

Example 2.12. To better illustrate how `break` and `continue` work, let's write a method that has no real utility, but demonstrates the utility of these keywords. Our method, namely `countOdds`, will count the number of odd integers generated by a random number generator from 1 to 100. Though, to make it more interesting, we will not count odd numbers less than 50, and if we ever generate a number greater than 95, we stop looping and return the count. The idea is to use `break` when we generate a number between 96 and 100, and to use `continue` to skip over incrementing the counter if we find a number less than 50. The `continue` keyword jumps program control to the top of the loop, skipping over any remaining statements in the body. Because this program relies solely on random chance, writing tests is not helpful. In the code segment, using an `else` statement would be preferred over the superfluous use of `continue`, but we demo it to at least portray its existence. The only way to exit the infinite `while` loop is to use either `break`, `return`, or stop the program some other way (e.g., through a program crash or forced termination).

Listing 2.50

```
class RandomNumbers {

  /**
   *
   * @return the number of odd values counted.
   */
  static int countOdds() {
    int cnt = 0;
    Random rng = new Random();
    while (true) {
      int r = 1 + rng.nextInt(100);
      if (r > 95) { break; }
      else if (r < 50 || r % 2 == 0) { continue; }
      cnt++;
    }
    return cnt;
  }
}
```

Example 2.13. Let's write one more example of a loop where we go directly from the problem statement to the code in question. Suppose we're writing a method that receives some string s, a "search string" f, and a replacement string r, and our goal is to replace all occurrences of f with r without helper methods in the `String` class, e.g., `indexOf`, `contains`, and certainly not `replace`. Of course, we start by writing the signature, documentation comments, and a handful of tests. We emphasize writing more tests than perhaps normal for this specific problem due to its complexity and number of "edge cases," i.e., cases in which a method might not account for because of their obscurity.

Listing 2.51—Find & Replace Tester

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

class FindReplaceTester {

  @Test
  void testFindReplace() {
    assertAll(
      () -> assertEquals("hiya there!",
                         replace("hiya there!", "", "replace!")),
      () -> assertEquals("hiya there!",
                         replace("hiya there!" "ya T", "replace!")),
      () -> assertEquals("Spaces_With_Underscores",
```

```
                      replace("Spaces With Underscores", " ", "_")),
      () -> assertEquals("heyyyyyay",
                      replace("heyaaaaay", "aa", "yy")),
      () -> assertEquals("hello",
                      replace("hillo", "i", "e")),
      () -> assertEquals("heyheyhey",
                      replace("hiyhiyhiy", "i", "e")),
      () -> assertEquals("We replaced the entire string!",
                      replace("Hello world, how are you?",
                              "Hello world, how are you?"
                              "We replaced the entire string!")));
  }
}
```

Now we write the implementation of `replace`, which breaks down into a few cases: while traversing, are did we find the start of the search string f, did we find the end of f, and did we encounter a break in f? Let's go from the start to the end in our analysis. When traversing over the input string s, if we find the character that starts f in s, we increment a counter c, and continue counting as long as the characters in the particular substring of s match the characters in f. As an example, in the string s = `"hello"`, if f = `"ell"`, once our traversal finds `"e"`, we increment c by one to designate that we matched one character of f. We continue this until we either hit the end of f, the end of s, or we find a non-matching character. In the former instance, we append the replacement string r onto our newly-constructed string s', because it indicates that we encountered a full match of f inside s, meaning it is of course replaced by r. In the second instance, we reach the end of s, meaning that we are out of characters to match. Therefore, we either copy over the last c characters from s into s' if we have not also hit the end of f, or r otherwise. For example, if s = `"hello"`, and f = `"low"`, we reach the end of s prior to finishing the substring f, meaning that we only copy over those final characters of s and do not append r. Conversely, if f = `"lo"` in that instance, we instead substitute the suffix `"lo"` with whatever is the value of r. Finally, when we encounter a non-matching character, we terminate the current replacement strategy and simply append the last c characters of s onto s' and move the index forward.

Listing 2.52—Find & Replace Implementation

```java
import java.lang.StringBuilder;

class FindReplace {

  /**
   * Searches through a string s for occurrences of "find", and
   * replaces them with "repl".
   * @param s string to search through.
   * @param find string to find.
   * @param repl string to replace "find" with.
   * @return new replaced string.
   */
  static String replace(String s, String find, String repl) {
    StringBuilder sb = new StringBuilder(s.length());
    for (int i = 0; i < s.length(); i++) {
      int pos = 0;
      int j;
      for (j = i; j < s.length(); j++) {
        // If we find the entire "find" string, append the replacement.
        if (pos >= find.length()) {
```

```
          sb.append(repl);
          i = j - 1;
          break;
        }
        // If we are in the middle of searching and we find a non-matching
        // character, append everything up until this point and break.
        else if (s.charAt(j) != find.charAt(pos)) {
          sb.append(s, i, i + pos + 1);
          i = j;
          break;
        }
        // If we are matching, continue to search.
        else if (s.charAt(j) == find.charAt(pos)) {
          pos++;
        }
      }
      // If we reach the end of the string and are in the middle of
      // searching, we append it to the sting.
      if (pos > 0 && j >= s.length()) {
        sb.append(repl);
        break;
      }
    }
    return sb.toString();
  }
}
```

Chapter Exercises

Exercise 2.1. (⋆)
Design the `boolean isStrictlyIncreasing(int x, int y, int z)` that determines if three integers x, y, and z are strictly increasing. By this, we mean that $x < y < z$.

Exercise 2.2. (⋆)
Design the `String numStuff(int n)` that determines if an integer n is greater than 100. If so, return the string of n divided by two. If the number is less than 50, return the string of n divided by five. In any other case, return the string "N/A."

Exercise 2.3. (⋆)
Design the `boolean canVote(int age)` that, when given an integer variable *age* in years, returns whether or not someone who is that age is able to legally vote in the United States. For reference, someone may legally vote once they turn eighteen years old.

Exercise 2.4. (⋆)
Design the `int max(int x, int y, int z)` that returns the maximum of three integers x, y, and z. Do not use any built-in (`Math` library) methods.

Exercise 2.5. (⋆)
Design the `int computeRoundSum(int x, int y, int z)` method that computes the sum of the rounded values of three integers x, y, and z. By "rounded values," we mean that we round their least significant digit, i.e., the rightmost digit either up or down depending on its value. For instance, 12 rounds down to 10, 59 rounds up to 60, and 1009 rounds up to 1010. If the number is negative, round towards zero when necessary, e.g., -7 rounds down to -10, -2 rounds up to zero.

Exercise 2.6. (⋆)
Design the `boolean lessThan20(int x, int y, int z)` method that, when given three integers x, y, and z, returns whether or not one of them is less than twenty away from another. For example, `lessThan20(19, 2, 412)` returns true because 2 is less than 20 away from 19. Another example is `lessThan20(999, 888, 777)`, which returns false because none of the numbers have a difference less than twenty.

Exercise 2.7. (⋆)
Design the `boolean canSleepIn(String d, boolean onVacation)` method, which determines whether or not someone can sleep in. Someone is able to sleep in if it is a weekend or they are on vacation. The input d is passed as a day of the week, e.g., `"Monday"`, ..., `"Sunday"`.

Exercise 2.8. (⋆)
Design the `boolean isEvenlySpaced(int x, int y, int z)` method, which receives three integers x, y, and z, and returns whether they are evenly spaced. Evenly spaced means that the difference between the smallest and medium number is the same as the difference between the medium and largest number.

Exercise 2.9. (⋆)
Design the `String cutTry(String s)` method, which receives a string s and, if s ends with `"try"`, it is removed. Otherwise, the original string is returned.

Exercise 2.10. (⋆)
Design the `String popChars(String s, char c, char d)` method, which receives

a string s and two characters c, d. The method removes c if s starts with c, and removes d if the second character of s is d. The remainder of the string is the same.

Exercise 2.11. ($\star$)
Design the `String middleString(String a, String b, String c)` method, which receives three strings a, b, and c, and returns the string that is "in between" the others in terms of their lexicographical content. You cannot sort the strings or use an array.

Exercise 2.12. ($\star$)
In propositional logic, there are several *connectives* that act on boolean truth values. These include logical conjunction $\wedge$, disjunction $\vee$, conditional $\rightarrow$, biconditional $\leftrightarrow$, and negation $\neg$. We can represent *schemata* as a series of composed method calls. For example, an evaluation of

$$`P \rightarrow \neg(Q \leftrightarrow \neg R)'$$

where 'P' and 'R' are assigned to `false` and 'Q' is assigned to `true`, is equivalent to

```
static final boolean P = false;
static final boolean Q = true;
static final boolean R = false;
cond(P, not(bicond(Q, not(R))))
```

The presented schema resolves to `true`.

Design methods for the five connectives according to the following truth tables. These methods should be called `cond`, `bicond`, `and`, `or`, and `not`. Assume that T is `true` and F is `false`.

P	$\neg P$
T	F
F	T

Truth Table of '$\neg P$'

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

Truth Table of '$P \wedge Q$'.

P	Q	$P \vee Q$
T	T	T
T	F	T
F	T	T
F	F	F

Truth Table of '$P \vee Q$'.

P	Q	$P \rightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

Truth Table of '$P \rightarrow Q$'.

P	Q	$P \leftrightarrow Q$
T	T	T
T	F	F
F	T	F
F	F	T

Truth Table of '$P \leftrightarrow Q$'.

Exercise 2.13. (⋆)
Design the `boolean isInsideCircle(double cx, double cy, double r, double px, double py)` method that, when given a circle centered at (c_x, c_y) and radius r as well as a point at (p_x, p_y), returns whether the point is located strictly inside the circle.

Exercise 2.14. (⋆)
Design the `boolean isInsideRectangle(double rx, double ry, double w, double h, double px, double py)` method that, when given a rectangle centered at (r_x, r_y), width w and height h as well as a point (p_x, p_y), returns whether the point is located strictly inside the rectangle.

Exercise 2.15. (⋆⋆)
Carlo is shipping out orders of candy to local grocery stores. Boxes have a maximum weight defined by a value w, and we can (potentially) fit both small and large bars of candy in a box. Design the `int fitCandy(int s, int l, int w)` method that, when given a number of small bars s, large bars l, and maximum weight w, determines the number of small candy bars he can fit in the box. Large bars weigh five kilograms, and small bars weigh one kilogram. Note that Carlo always tries to fit large candies first before small. Return `-1` if it is impossible to fill the box with the given criteria. Below are some test examples. Hint: consider this as an analysis of three cases.

```
fitCandy(4, 1, 9)      => 4
fitCandy(4, 1, 4)      => 4
fitCandy(1, 2, 6)      => 1
fitCandy(6, 1, 13)     => -1
fitCandy(60, 100, 550) => 50
fitCandy(7, 1, 12)     => 7
fitCandy(7, 1, 13)     => -1
```

Exercise 2.16. (⋆⋆⋆)
An IPv4 address contains four integer values stored in four octets, separated by dots. For instance, `192.168.1.244` is a valid IPv4 address. Another example is `149.165.192.52`. Design the `boolean isValidIpv4(String ip)` method that, when given a string, determines whether or not it represents a valid IPv4 address. Each octet must be an integer between zero and 255 inclusive. Note that some IPv4 addresses are, in reality, nonsensical, e.g., `0.0.0.0`, but we will not consider these as invalid. Below the examples is a helper method, `isNumeric`, to determine whether or not a string is "numeric." Understanding *how* this helper method works is unimportant for the time being. You **cannot** use arrays, loops, or regular expressions to solve this problem. Finally, you will need to use `Integer.parseInt`, `substring`, and `indexOf`.

```
isValidIpv4("192.168.1.244")    => true
isValidIpv4("149.165.192.52")   => true
isValidIpv4("192.168.1.256")    => false
isValidIpv4("192.168.1201.23")  => false
isValidIpv4("192.168.1201.ABC") => false
isValidIpv4("ABC.DEF.GHI")      => false
isValidIpv4("192.168.1A6.201")  => false
```

```
/**
 * Determines whether or not we can convert a given string into
 * an integer datatype.
 * @param n - input string.
 * @return true if the string is convertable to an integer, false otherwise.
```

```
 */
static boolean isNumeric(String n) {
  try {
    Integer.parseInt(n);
    return true;
  } catch (NumberFormatException ex) {
    return false;
  }
}
```

Exercise 2.17. (⋆)
Design the `String stateOfMatter(double t, char u)` method that receives a water temperature as a `double` and a unit as a `char`, i.e., either `'C'` or `'F'` for Celsius and Fahrenheit respectively. Return a string representing whether the water is a liquid, solid, or gas at sea level.

Exercise 2.18. (⋆)
Design the `double computeGpa(String grade)` method that translates a letter grade into a number grade. Letter grades are A, B, C, D, and F, possibly followed by + or -. Their numeric values are 4, 3, 2, 1, and 0. There is no F+ or F-. A+ increases the numeric value by 0.3, a - decreases it by 0.3. However, an A+ has value 4.0.

Exercise 2.19. (⋆)
Design the `String sortThreeStrings(String a, String b, String c)` method that receives three strings and sorts them lexicographically. Return the sorted set of strings as a string itself, separated by commas. For example, if the input is `Charlie`, `Able`, `Baker`, you should return `Able,Baker,Charlie`.

Exercise 2.20. (⋆)
A year with 366 days is called a leap year. Leap years are necessary to keep the calendar synchronized with the sun because the earth revolves around the sun once every 365.25 days. Actually, that figure is not entirely precise, and for all dates after 1582 the Gregorian correction applies. Usually years that are divisible by 4 are leap years, for example 1996. However, years that are divisible by 100 (for example, 1900) are not leap years, but years that are divisible by 400 are leap years (for example, 2000). Design the `boolean isLeapYear(int y)` method that receives a year (as an integer) y and computes whether y is a leap year. Use a single `if` statement and Boolean operators.

Exercise 2.21. (⋆)
Design the `double computeDiscount(double c, int age, boolean isStudent)` method that computes a discount for some item cost c based on their age age and student status according to the following criteria:

- If $age < 18$, apply a 20% discount.
- If $18 \leq age \leq 25$ and they are a student, apply a 25% discount. If they are not a student, do not apply a discount.
- If $age \geq 65$ and they are a student, apply a 30% discount. If they are not a student, apply a 15% discount.
- All other cases should not have a discount applied.

Your method should return the total cost of the item after applying the discount.

Exercise 2.22. (⋆)
Design the `double computeTaxCost(double itemCost, String state)` method that computes the tax for some item based on the state in which it is purchased. The method should return the total cost of the item, which includes the taxed amount. The tax rates are as follows:

- `"CA"`: 9.25%
- `"NY"`: 4.0%
- `"NC"`: 6.625%
- `"SC"`: 6.0%
- `"VA"`: 6.25%
- `"WA"`: 6.5%
- `"IN"`: 8.0%

Exercise 2.23. (⋆)
Design the recursive `int countStr(String s)` that counts the number of times the substring `"str"` appears in a given string s.

Exercise 2.24. (⋆)
Rewrite the `countStr` method to use tail recursion. Name this new version of the method `countStrTR`. Hint: you will need to design a `private static` helper method. The `countStrTR` method should only have one parameter.

Exercise 2.25. (⋆)
Design the recursive `String replaceAB(String s)` method that replaces any occurrence of the character `'A'` with the character `'B'` in a given string s.

Exercise 2.26. (⋆)
Rewrite the `replaceAB` method to use tail recursion. Name this new version of the method `replaceABTR`. Hint: you will need to design a `private static` helper method. The `replaceABTR` method should only have one parameter.

Exercise 2.27. (⋆)
Elephants have two ears, right? Design the recursive `int countElephantEars(int n)` that returns the total number of elephant ears that are in a group of n elephants.

Exercise 2.28. (⋆)
Rewrite the `countElephantEars` method to use tail recursion. Name this new version of the method `countElephantEarsTR`. Hint: you will need to design a `private static` helper method. The `countElephantEarsTR` method should only have one parameter.

Exercise 2.29. (⋆)
This question has two parts.

(a) Design the `raiseLowerToUpperTR` tail recursive method, which receives a string and returns the number of lowercase characters raised to the number of uppercase characters, ignoring any other character. If there are no lowercase or uppercase characters, return zero. Hint: you will need to design a `private static` helper method to solve this problem.

(b) Design the `raiseLowerToUpperLoop` method that solves this problem using a loop.

If you write tests for one of these methods, you should be able to propagate it through the other, so write plenty!

Exercise 2.30. ($\star\star$)
This question has two parts.

(a) Design the `isPrimeTR` tail recursive method, which receives a positive integer and determines if it is prime. Recall that a number is prime if and only if it evenly divides only one and itself. Hint: you will need to design a `private static` helper method to solve this problem.

(b) Design the `isPrimeLoop` method that solves the problem using a loop.

If you write tests for one of these methods, you should be able to propagate it through the other, so write plenty!

Exercise 2.31. ($\star$)
This question has two parts.

(a) Design the `isPalindromeTR` tail recursive method, which receives a string and determines if it is a palindrome. Recall that a palindrome is a string that is the same backwards as it is forwards. E.g., “racecar.” **Do not** use a (character) array, `StringBuilder`, `StringBuffer`, or similar, to solve this problem. It *must* be naturally recursive.

(b) Design the `isPalindromeLoop` method that solves the problem using a loop. The same restrictions from the previous problem hold true for this one.

If you write tests for one of these methods, you should be able to propagate it through the other, so write plenty!

Exercise 2.32. ($\star$)
This question has two parts.

(a) Design the `gcdTR` tail recursive method, which receives two integers and returns the greatest common divisor between the two. Euclid’s algorithm is the basis for this approach and is a tail recursive algorithm by design.

(b) Design the `gcdLoop` method that solves the problem using a loop.

If you write tests for one of these methods, you should be able to propagate it through the other, so write plenty!

Exercise 2.33. ($\star$)
This question has two parts.

(a) Design the `isNestedParenthesesTR` tail recursive method, which receives a string and determines if its parentheses pairs are “balanced.” A pair of parentheses is balanced if it is a nesting of zero or more pairs of parenthesis, like “(())” or “((()))”. Note that pairs like “(()())” will not be tested.

(b) Design the `isNestedParenthesesLoop` method that solves the problem using a loop.

If you write tests for one of these methods, you should be able to propagate it through the other, so write plenty!

Exercise 2.34. ($\star$)
This question has three parts.

(a) The *hyperfactorial* of a number, namely $H(n)$, is the value of $1^1 \cdot 2^2 \cdot \ldots \cdot n^n$. As you might imagine, the resulting numbers from a hyperfactorial are outrageously large. Therefore we will make use of the `long` datatype rather than `int` for this problem.

Design the standard recursive `hyperfactorial` method, which receives a long integer n and returns its hyperfactorial.

(b) Then, design the `hyperfactorialTR` method that uses tail recursion and accumulators to solve the problem. Hint: you will need to design a `private static` helper method to solve this problem.

(c) Finally, design the `hyperfactorialLoop` method that solves the problem using a loop.

If you write tests for one of these methods, you should be able to propagate it through the rest, so write plenty!

Exercise 2.35. ($\star$)
This question has three parts.

(a) The *subfactorial* of a number, namely $!n$, is the number of permutations of n objects such that no object appears in its natural spot. For example, take the collection of objects $\{a, b, c\}$. There are 6 possible permutations (because we choose arrangements for three items, and $3! = 6$): $\{a, b, c\}, \{a, c, b\}, \{b, c, a\}, \{c, b, a\}, \{c, a, b\}, \{b, a, c\}$, but only two of these are *derangements*: $\{b, c, a\}$ and $\{c, a, b\}$, because no element is in the same spot as the original collection. Therefore, we say that $!3 = 2$. We can describe subfactorial as a recursive formula:

$$
\begin{aligned}
!0 &= 1 \\
!1 &= 0 \\
!n &= (n-1) \cdot (!(n-1) + !(n-2))
\end{aligned}
$$

Design the standard recursive `subfactorial` method, which receives an long integer n and returns its subfactorial. Because the resulting subfactorial values can grow insanely large, we will use the `long` datatype instead of `int`.

(b) Then, design the `subfactorialTR` method that uses tail recursion and accumulators to solve the problem. Hint: you will need to design a `private static` helper method to solve this problem.

(c) Finally, design the `subfactorialLoop` method that solves the problem using a loop.

If you write tests for one of these methods, you should be able to propagate it through the rest, so write plenty!

Exercise 2.36. ($\star$)
This question has three parts.

(a) Design the standard recursive `collatz` method, which receives a positive integer and returns the Collatz sequence for said integer. This sequence is defined by the following recursive process:

`collatz(1)` = `1`
`collatz(n)` = `collatz(3 * n + 1)` if `n` is odd.
`collatz(n)` = `collatz(n / 2)` if `n` is even.

The sequence generated is the numbers received by the method until the sequence reaches one (note that it is an open research question as to whether this sequence converges to one for every positive integer). So, `collatz(5)` returns the following `String` of comma-separated integers: `"5,16,8,4,2,1"`. **The last number cannot have a comma afterwards**.

(b) Then, design the `collatzTR` method that uses tail recursion and accumulators to solve the problem. Hint: you will need to design a `private static` helper method to solve this problem.

(c) Finally, design the `collatzLoop` method that solves the problem using a loop.

If you write tests for one of these methods, you should be able to propagate it through the rest, so write plenty!

Exercise 2.37. (⋆)
This question has three parts.

(a) Design the standard recursive `parenthesesDepth` method, which receives a string of parentheses and returns an integer representing the "depth" at the end of the string. Each instance of `(` increments the depth counter, and each instance of `)` decrements the depth counter. So, e.g., `"((())()))))(()"` is $1+1+1-1-1+1-1-1-1-1-1+1+1=-2$.

(b) Then, design the `parenthesesDepthTR` method that uses tail recursion and accumulators to solve the problem. Hint: you will need to design a `private static` helper method to solve this problem.

(c) Finally, design the `parenthesesDepthLoop` method that solves the problem using a loop.

If you write tests for one of these methods, you should be able to propagate it through the rest, so write plenty!

Exercise 2.38. (⋆)
This question has three parts.

(a) Design the standard recursive `countdown` method, which receives an `int` $n \geq 0$ and returns a `String` containing a sequence of the even numbers from n down to 0 inclusive, separated by commas.

(b) Then, design the `countdownTR` method that uses tail recursion and accumulators to solve the problem. Hint: you will need to design a `private static` helper method to solve this problem.

(c) Finally, design the `countdownLoop` method that solves the problem using a loop.

If you write tests for one of these methods, you should be able to propagate it through the rest, so write plenty!

Exercise 2.39. (⋆⋆)
This question has three parts.

(a) Design the standard recursive `chickenCounter` method, which receives a string s and returns the number of times the substring `"chicken"` appears in s. You must account for overlapping instances of `"chicken"`. For example, calling the method with `"abcchickechickenn"` returns 2 because, after removing the substring `"chicken"` from the original string, we are left with `"abcchicken"`, which itself contains another instance of `"chicken"`.

(b) Then, design the `chickenCounterTR` method that uses tail recursion and accumulators to solve the problem. Hint: you will need to design a `private static` helper method to solve this problem.

(c) Finally, design the `chickenCounterLoop` method that solves the problem using a loop.

If you write tests for one of these methods, you should be able to propagate it through the rest, so write plenty!

Exercise 2.40. (⋆⋆⋆)
This question has five parts. We need to provide some background for the question first. An *encoded string* S is one of the form:

$$S = G^*$$
$$G = NL$$
$$N = [0\text{--}9]^+$$
$$L = [\text{a--z}]^+$$

We imagine this didn't clear up what the definition means. Take the encoded string `"3[a]2[b]"` as an example. The resulting decoded string is `"aaabb"`, because we create three copies of `"a"`, followed by two copies of `"b"`. Another example is `"4[abcd]"`, which returns the string containing `"abcdabcdabcdabcd"`.

(a) First, design the `int retrieveN(String s)` that returns the integer at the start of an encoded string. Take the following examples as motivation.

```
retrieveN("3[a]2[b]")      => 3
retrieveN("47[abcd]")      => 47
retrieveN("1[bbbbb]3[a]") => 1
```

(b) Next, design the `String cutN(String s)` method that returns a string without the integer at the start of an encoded string. Hint: use `indexOf` and `substring`.

```
cutN("3[a]2[b]")      => "[a]2[b]
cutN("47[abcd]")      => "[abcd]"
cutN("1[bbbbb]3[a]") => "[bbbbb]3[a]"
```

(c) Design the *standard recursive* `decode` method, which receives an encoded string and performs a decoding operation.

(d) Design the `decodeTR` and `decodeTRHelper` methods. The former acts as the driver to the latter; the latter solves the same problem that `decode` does, but it instead uses tail recursion. Remember to include the relevant access modifiers!

(e) Design the `decodeLoop` method, which solves the problem using either a `while` or `for` loop.

Exercise 2.41. (⋆⋆)
Design the `boolean isNumberPalindrome(int n)` method that, when given an integer n, returns whether or not that number is a palindrome. You cannot convert the number to a string.

Exercise 2.42. (⋆⋆⋆)
The C programming language contains the `atoi` "ascii-to-integer" function, which receives a string and, if the string represents some integer, returns the number converted to an integer. Design the `int atoi(String s)` method that, when given a string s, returns its value as an integer if it can be parsed as an integer. When parsing the integer, ignore all leading zeroes and leading non-digits. Upon finding the first non-zero digit, if it exists, begin interpreting the string as a number. At any point thereafter, if a non-digit is encountered, return the number parsed up to that point. The given integer can also contain a sign, e.g., $+$ or $-$. If the value exceeds the bounds of an integer (i.e., `Integer.MAX_VALUE` or `Integer.MIN_VALUE`), return zero. Writing enough tests is *crucial* to getting this correct! We provide some examples below.

```
atoi("ABCD")            => 0
atoi("42")              => 42
atoi("000042")          => 42
atoi("004200")          => 4200
atoi("ABCD42ABCD)       => 42
atoi("ABCD-+42ABCD)     => 42
atoi("ABCD+-42ABCD)     => -42
atoi("9999999999999")   => 0
atoi("-9999999999999")  => 0
atoi("000-42000")       => -42000
atoi("000-ABCD)         => 0
atoi("000+42ABCD)       => 42
atoi("8080*8080)        => 8080
```

Exercise 2.43. (⋆⋆)
Design the `int indexOf(String s, String k)` method, which receives two strings s, k, and returns the first index of k in s. Note that k may be any arbitrary string and not just a a single character. If k is not in s, return `-1`. You **cannot** use the `indexOf` method provided by the `String` class.

Exercise 2.44. (⋆⋆)
Design the `String substring(String s, int a, int b)` method, which receives a string and two integers a, b, and returns the substring between these indices. If either are out of bounds of the string, return `null`. You **cannot** use the `substring` method(s) provided by the `String` class.

Exercise 2.45. (⋆)
Design the `boolean isEqualTo(String s1, String s2)` method, which receives two strings s_1 and s_2, and determines whether they are lexicographically equal. You cannot use the built-in `equals` or `compareTo` methods.

Exercise 2.46. (⋆⋆)
Design the `int compareTo(String s1, String s2)` method that receives two strings s_1 and s_2, and compares their contents lexicographically. If `s1` is less than `s2`, return `-1`. If `s1` is greater than `s2`, return `1`. Otherwise, return `0`. Note that our implementation of `compareTo` will differ from Java's in that, if `s1` has less characters than `s2`, we return `-1`; if `s2` has less characters than `s1`, we return `1`. Otherwise, we do the character-by-character comparison.

Exercise 2.47. (⋆⋆)
Design the `String trim(String s)` method, which receives a string s and a character

ch, and returns a string with all leading and trailing occurrences of ch removed. For instance, trim("aaHelloa", 'a') returns "Hello". Hint: while you cannot use Java's substring method, you can certainly use the one you wrote previously to solve this problem!

Exercise 2.48. (⋆)
Design the String trimSpace(String s) method, which receives a string s and returns a new string with all leading and trailing spaces removed. You cannot use the trim method provided by the String class.

Exercise 2.49. (⋆⋆)
Design the boolean containsMiddleABC(String s) method, which receives a string s and returns whether s contains the substring "ABC" in the "middle." We define the "middle" as the point where number of characters on the left and right differ by at most one. You **cannot** use any String methods to solve this problem except .length and .charAt.

```
assertTrue(middleABC("helloABChiya!"));
assertTrue(middleABC("ABC"));
assertTrue(middleABC("aABCc!"));
assertFalse(middleABC("notInTheMiddleABCmid!"));
```

Exercise 2.50. (⋆⋆⋆)
Design the String censor(String s, String c) method, which receives a string s and another string c. It should return a "censored" version of s, wherein each instance of c in s is replaced by asterisks. You **cannot** use any String methods to solve this problem except .length and .charAt. This problem is harder than it looks due to these limitations.

Exercise 2.51. (⋆)
Design the boolean isSelfDividing(int n) method, which receives an integer n and returns whether the sum of its digits evenly divide n. You must perform the arithmetic manually; you **cannot** convert the value to a String or use an array.

Exercise 2.52. (⋆)
Design the boolean allSelfDividing(int n) method, which receives an integer n and returns whether each digit evenly divides n. If any digit is zero, then return false. You must perform the arithmetic manually; you **cannot** convert the value to a String or use an array.

Exercise 2.53. (⋆⋆)
Design the int strSumNums(String s) that computes the sum of each *positive integer* (≥ 0) in a string s. See the below test cases for examples. You may assume that each integer in s, should there be any, is in the bounds of a positive int, i.e., 0 and $2^{31} - 1$. Hint: use Character.isDigit to test whether a character c is a digit, and Integer.parseInt to convert a String to an int.

```
assertEquals(100, strSumNums("hello50how20are30you?"));
assertEquals(10, strSumNums("t1h1i1s1i1s1e1a1s1y1!"));
assertEquals(0, strSumNums("there are no numbers :("));
assertEquals(0, strSumNums("still 0 just 0 zero0!"));
assertEquals(500000, strSumNums("500000"));
```

Exercise 2.54. (⋆⋆⋆)
Design the String stripComments(String s) method that, when given a string s rep-

resenting a (valid) Java program, returns a string where all comments (single-line, multi-line, and Java documentation) have been removed. You **cannot** use any `String` helper methods (e.g., `strip`, `split`) to solve this problem nor can you use regular expressions.

Exercise 2.55. (⋆)
Design the `double approxPi(int n)` that approximates π using the following formula:

$$\pi = 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} - \frac{1}{15} + \frac{1}{17} - \cdots\right)$$

That is, given an input number n, compute that many terms of the above sequence. Return the difference between this approximation and Java's built-in `Math.PI` using `Math.abs`.

Exercise 2.56. (⋆)
Design the `String toBinary(int n)` that converts a positive integer n into a `String` that represents its binary counterpart. We present some examples below.

```
toBinary(13)  => "1101"
toBinary(144) => "10010000"
toBinary(25)  => "11001"
```

Exercise 2.57. (⋆⋆)
Design the `String toBase(int n, int u, int v)` method that converts a positive integer n_u in base u to base v, as a `String`. You may assume that $2 \leq u, v \leq 16$. Any base above ten uses A, B, ..., F for 11, 12, ..., 15 respectively. Hint: multiply when going down in bases ($v < u$), divide when going up ($v > u$).

Exercise 2.58. (⋆)
Design the `int countPairs(int n)` method that computes the number of pairs (a, b) that satisfy the equation $(a^2 + b^2 + 1)/(ab)$ such that $1 \leq a \leq b < n$. To "satisfy the equation," in this context, means that the quotient is an integer.

Exercise 2.59. (⋆)
Design the `String mirrorEnds(String s)` method that, when given a string s, looks for a mirror image (backwards) string at both the beginning and end of the given string. In other words, zero or more characters at the very beginning of the given string, and at the very end of the string in reverse order (possibly overlapping). For example, the string `"abXYZba"` has the mirror end `"ab"`. If there is no such string, return `null`.

```
mirrorEnds("abXYZba") => "ab"
mirrorEnds("abca")    => "a"
mirrorEnds("aba")     => "aba"
mirrorEnds("abc")     => null
```

Exercise 2.60. (⋆⋆)
Design the `String multTable(int a, int b)` method that, when given two integers a and b such that $a \leq b$, computes the "multiplication table" from a to b. We provide some test cases below. Note that the newline is just for formatting purposes.

```
multTable(3, 3) => "1*1=1,1*2=2,1*3=3,2*1=2,2*2=4,
                    2*3=6,3*1=3,3*2=6,3*3=9"
multTable(2, 6) => "1*1=1,1*2=2,1*3=3,1*4=4,1*5=5,
                    1*6=6,2*1=2,2*2=4,2*3=6,2*4=8,
                    2*5=10,2*6=12"
```

Exercise 2.61. (⋆⋆⋆)
The *definite integral* of a function f, defined as $\int_a^b f(x)\,\mathrm{d}x$, produces the area under the

curve of f on the interval $[a, b]$. The thing is, though, integrals are defined in terms of *Riemann summations*, which provide estimations on the area under a curve. Riemann sums approximate the area by creating rectangles of a fixed width Δ, as shown in 2.7 for an arbitrary function f. Left-Riemann, right-Riemann, and midpoint-Riemann approximations define the focal point, i.e., the height, of the rectangle. Notice that, in Figure 2.7, we use a midpoint-Riemann sum with $\Delta = 0.2$, in which the collective sum of all the rectangle areas is the Riemann approximation. Your job is to use this idea to approximate the area of a circle.

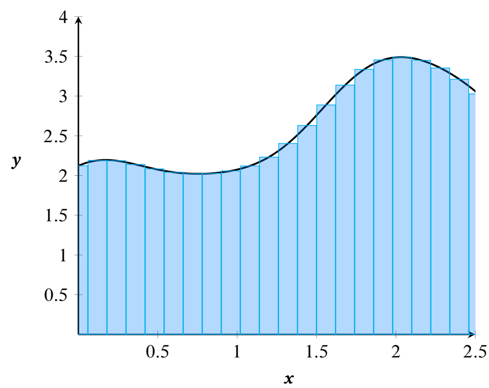


Figure 2.7: Midpoint-Riemann Approximation of a Function

Design the `double circleArea(double r, double delta)` method, which receives a radius r and a delta Δ. It computes (and returns) a left/right-Riemann approximation of the area of a circle. Hint: if you compute the left/right-Riemann approximation of one quadrant, you can very easily obtain an approximation of the total circle area. We illustrate this hint in Figure 2.8 where $\Delta = 0.5$ and its radius $r = 2$. Note that the approximated area will vary based on the chosen Riemann approximation.[1] Further note that no calculus knowledge is necessary to solve this exercise.

[1] A left-Riemann sum under-approximates the area, whereas a right-Riemann sum provides an over-approximation. A midpoint approximation uses the average between the left and right approximations. It should be noted that, in the general case, these statements do not hold as they depend on the interval we integrate our function over.

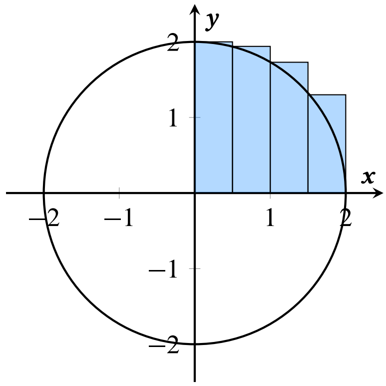


Figure 2.8: Left-Riemann Approximation of a Function

Exercise 2.62. (⋆⋆⋆)
Speech-to-text software plays a significant role in accessibility for those who may not be able to type quickly or at all. Design the `String speechToText(String s)` method that, when given a "speech string', returns the corresponding text. A speech string, in this context, is a string spoken, in English, by a person, which may or may not contain punctuation. If a speech string contains a word that represents punctuation, e.g., `"period"`, `"question mark"`, encode this punctuation in the returned text. For example, `speechToText("hello period how are you question mark")` returns the string `"Hello. How are you?"`. You should also account for quotations, e.g., `speechToText-("hello quote how are you question mark unquote")` returns the string `Hello. "how are you?"`.

3. Arrays, Collections, and Generics

3.1 Arrays

Thus far all of our work has been with data available as it is received by a method. That is, when we invoke a method with some data values, we have access to them at that point in time. *Arrays* allow us to store values, similar to how we use variables, but for an arbitrary/indeterminate number of values.

Arrays store *elements* and *indices* of some type. An element is just a value in an array. The *index* of an element is its location in the array. Indices of an array are indexed from zero, much like strings. Thus, the first element of an array is located at index zero, whereas the last element is located at the index $|A|-1$, where $|A|$ denotes the number of elements, or *cardinality*, of some array A.

We store contiguous elements in arrays, all of the same type. This means that, if we declare an array of type `int`, we cannot store, say, a `String` in the array. We can declare an array variable using initializer lists:

```
int[] array = {5, 10, 15, 20, 100, 50};
```

To retrieve the size of an array, i.e., the number of elements it stores, we access the *.length* field of the array; e.g., `array.length`. For our example array, we see that its size is six. Moreover, `array[0]` stores `5`, and `array[5]` stores `50`. Accessing a negative index or beyond the bounds of the length results in an array index out of bounds exception. That is, accessing `array[-1]` or `array[6]` crashes the program. A common mistake is to access the index at the length of the array to retrieve the last element. Doing so represents a misunderstanding of how arrays compute indices of elements.

To declare an array of some type T, called A, that stores N elements, we write the following:

```
T[] A = new T[N];
```

We can store a value e at some arbitrary index i of array A, in addition to accessing the value at some index.

```
// Store "e" at index i of A.
A[i] = e;
```

Java Arrays

An *array* stores a fixed-size collection of elements of some type.

$T[]\ A =$ **new** $T[n]$ creates an array of type T, named A, that stores n elements.

$A[i]$ retrieves the element at index i^{th} of A. We refer to this as position $i+1$.

$A[i] = v$ assigns v to index i of A.

A**.length** returns the number of elements that the array can store.

Arrays.equals (A_1, A_2) returns whether or not the elements of A_1 are equal to the elements of A_2.

Arrays.toString (A) returns a string representation of the elements in A, separated by commas and enclosed by brackets.

Arrays.fill (A, v) populates A with v in every index.

Arrays.copyOf (A, n) returns a new array A' of the same type with the new size, padding with the necessary default elements or truncating.

Arrays.sort (A) performs an in-place sort of A, meaning the contents of A are modified.

Figure 3.1: Useful Array Methods.

```
// Print out the value at A[i].
System.out.println(A[i]);
```

Example 3.1. Let us declare an array A that stores the integers from zero to one hundred, in increments of ten.

```
int[] A = {0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
```

This is verbose and requires us to explicitly specify each individual constant. A better solution is to initialize the array to a size and populate its elements using a loop.

```
final int SIZE = 11;
int[] A = new int[SIZE];
for (int i = 0; i < A.length; i++) { A[i] = i * 10; }
```

We use i to iterate over the possible indices of our array. Before we explained that i is used out of standard convention, but we now say that i, in general, stands for either "iteration" or "index." We assign, to A, the value of i multiplied by ten. We can convert the array to a `String` using a utility method from the `Arrays` class (note the plural!); a "string-ified" array separates each element by commas and surrounds them with braces.

```
String s1 = Arrays.toString(A);
s1 => {0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100}
```

Example 3.2. Let us write a method that receives an array of `double` values and returns the sum of the elements. Of course, we need tests!

Listing 3.1—Sum of Double Array Tests

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

```java
class SumOfDoubleArrayTester {

  @Test
  void sumOfDoubleArrayTest() {
    assertAll(
      () -> assertEquals(0.0, sumOfDoubles(new double[]{})),
      () -> assertEquals(50.0, sumOfDoubles(new double[]{50.0})),
      () -> assertEquals(100.0, sumOfDoubles(new double[]{25.0, 50.0, 25.0})));
  }
}
```

Our method uses a local variable to accumulate the "running sum," so to speak, of the values seen so far from the given array.

Listing 3.2—Sum of Double Array Implementation

```java
class SumOfDoubleArray {

  /**
   * Computes the sum of the values in a double array.
   * @param double [] array of double values.
   * @return sum of those values.
   */
  static double sumOfDoubles(double[] arr) {
    double sum = 0;
    for (int i = 0; i < arr.length; i++) { sum += arr[i]; }
    return sum;
  }
}
```

Even though our code works and the tests that we wrote pass without question, there is a bit of verbosity with our loop; all we ever use the iteration variable, i, is for accessing a variable. In such circumstances, we may prefer using the enhanced `for` loop, which abstracts away the index and provides an iteration construct for accessing elements sequentially.

Listing 3.3—Sum of Double Array Enhanced `for` Loop

```java
class SumOfDoubleArray {

  static double sumOfDoubles(double arr) {
    double sum = 0;
    for (double e : arr) { sum += e; }
    return sum;
  }
}
```

Why might someone want to use the enhanced `for` loop over a standard `for`? In general, when we only want to access the elements themselves and not care about their position, the enhanced counterpart is favored; not having to concern ourselves with indices completely removes the possibility of accessing an out-of-bounds index.

***Example* 3.3.** Let us write a method that returns the largest integer in an array of integers. This is straightforward, but can be a little tricky to get right due to how we determine the "largest." Some programmers may choose to declare a value `largest` and assign it, say, -1, then if we encounter a larger value, overwrite its value. This works well when the provided array contains only positive values, but what if our array contains only negative numbers? This approach falls apart in such instances. To simplify the implementation, we will assume that the given array contains at least one value.

Listing 3.4—Largest Integer in Array Tests

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

class LargestIntTester {

  @Test
  void largestIntTester() {
    assertAll(
      assertEquals(4, largestInt(new int[]{4})),
      assertEquals(13, largestInt(new int[]{12, 13, 10, 9})),
      assertEquals(-5, largestInt(new int[]{-5, -7, -1932, -6, -6})),
      assertEquals(9, largestInt(new int[]{9, 9, 9, 9, 9, 9, 9, 9})),
      assertEquals(0, largestInt(new int[]{-321, -43, 0, -43, -321})),
      assertEquals(0, largestInt(new int[]{-9, 0, -8, -7, -1234})));
  }
}
```

Listing 3.5—Largest Integer in Array Implementation

```
class LargestInt {

  static int largestInt(int[] arr) {
    int max = arr[0];
    for (int i = 1; i < arr.length; i++) {
      if (arr[i] > max) { arr[i] = max; }
    }
    return max;
  }
}
```

A slightly more efficient and compact solution would be to wrap the conditional inside a call to `Math.max`, since the logic located within is effectively identical: `max = Math.max(arr[i], max)`.

Java arrays are rather primitive compared to other more-complex data structures.[1] The `Arrays` class provides a few convenient methods for working with arrays directly, but for the most part, arrays are used as the backbone of other data structures. In general, we use arrays when we want constant access times for elements. That is, if we know the index of an element e, we retrieve it using the aforementioned array bracket syntax. This idea holds true for modifying elements. If we want to know whether an array contains a value, we need to write our own method to check.

Example 3.4. Suppose we want to write a method that returns the index of an element e of an array of `String` values S. Doing so is straightforward: check each element, one by one, until we find the desired element, or return -1. Note the similarity to the `indexOf` method, which is part of the `String` class. To gain practice using both recursion and iteration, we will write three versions of this method: one using standard recursion, the second using tail recursion, and the third using a loop. Our standard recursive method will recurse on the length of the array minus one until it is a negative number. The tail

[1]Do not conflate this use of the "primitive" term with its use in describing "primitive datatypes."

recursive method is almost identical to the former, with the exception that upon finding the element, we return the index itself rather than relying on the recursion to unwind. The tests for these two are both trivial to design.

Listing 3.6—Standard Recursive `indexOf` Method Tester

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

class ArrayFinderTester {

  @Test
  void arrayFinderRecursiveTest() {
    String[] arrS = new String[]{"Hello", "hi", "hiya", "howdy", "hello"};
    assertAll(
      () -> assertEquals(2, indexOf(arrS, "hiya")),
      () -> assertEquals(0, indexOf(arrS, "Hello")),
      () -> assertEquals(4, indexOf(arrS, "hello")),
      () -> assertEquals(-1, indexOf(arrS, "ahoy")));
  }
}
```

Listing 3.7—Tail Recursive `indexOf` Method Tester

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

class ArrayFinderTailRecursiveTester {

  @Test
  void arrayFinderTailRecursiveTest() {
    String[] arrS = new String[]{"Hello", "hi", "hiya", "howdy", "hello"};
    assertAll(
      () -> assertEquals(2, indexOfTR(arrS, "hiya")),
      () -> assertEquals(0, indexOfTR(arrS, "Hello")),
      () -> assertEquals(4, indexOfTR(arrS, "hello")),
      () -> assertEquals(-1, indexOfTR(arrS, "ahoy")));
  }
}
```

Funnily enough, the standard recursive solution uses an accumulator like its tail recursive counterpart. Unfortunately, the standard recursive method is unnecessarily complicated and we would not recommend using it for searching through an array. Its implementation is here only for pedagogical purposes. On the other hand, the tail recursive method recurses over the accumulator, which serves as the current index to investigate. If this value exceeds the bounds of the array, we return -1. If $S[i]$ is equal to k, we return i. Otherwise, we recurse and increment i by one.

Listing 3.8—Searches an Array using Standard Recursion

```
class ArrayFinder {

  static String indexOf(String[] arrS, String k) {
    return indexOf(arrS, k, 0);
  }
```

```
  private static String indexOfHelper(String[] arrS, String k, int i) {
    if (i >= arrS.length) { return -1; }
    else if (arrS[i].equals(k)) { return 0; }
    else { return 1 + indexOfHelper(arrS, k, i + 1); }
  }
}
```

Listing 3.9—Searches an Array using Tail Recursion

```
class ArrayFinder {

  static String indexOfTR(String[] arrS, String k) {
    return indexOf(arrS, k, 0);
  }

  private static String indexOfTRHelper(String[] arrS, String k, int i) {
    if (i >= arrS.length) { return -1; }
    else if (arrS[i].equals(k)) { return i; }
    else { return indexOfHelper(arrS, k, i + 1); }
  }
}
```

In converting the tail recursive solution to use iteration, we will make use of the translation pipeline. Our base case is when i equals or exceeds the length of the array, so the negated expression is our loop condition. Moreover, we can place a conditional statement inside the loop, which returns whether $S[i]$ equals k for that value of i and, if so, we return i. We might also form a conjunction between the two conditions, whose exit condition is when one of those conditions is falsified. Because we have two different atomic return values, though, we will use the former approach. Because the test cases are, verbatim, those that we wrote for the recursive solutions, we omit them out of a desire for conciseness.

Listing 3.10—Searches an Array using a `while` Loop

```
class ArrayFinder {

  static String indexOfLoop(String[] arrS, String k) {
    int i = 0;
    while (!(i >= arrS.length)) {
      if (arrS[i].equals(k)) {
        return i;
      }
    }
    return -1;
  }
}
```

Of course, the conventional solution to this problem, especially since we know the upper bound on the number of iterations, would be to use a `for` loop. Doing so localizes the accumulator variable. Moreover, we *could* use the translation pipeline conditional expression, but it is idiomatic to loop while the index is less than the length of the array and use an expression describing this relationship.

Listing 3.11—Searches an Array using a `for` Loop

```
class ArrayFinder {
```

```
  static String indexOfLoop(String[] arrS, String k) {
    for (int i = 0; i < arrS.length; i++) {
      if (arrS[i].equals(k)) {
        return i;
      }
    }
    return -1;
  }
}
```

We are on our way to understanding the full signature of the `main` method. Now that we have covered arrays, we know what the `String[] args` parameter represents, but *why* it receives that array of strings remains a mystery. We can compile Java files using the terminal and the `javac` command. Moreover, when executing a Java file, we may pass to it *terminal arguments*, which are values that the program might use to configure settings or other miscellaneous information.

Example 3.5. Suppose we want to write a program, using the `main` method and terminal arguments, that performs an arithmatic operation on a collection of integers values, e.g., $5+3+17$. Additionally, we might want to let the user pass *flags* to denote these different operations, such as `--add` for addition, `--sub` for subtraction, and so on. So the user is not confused, we might also provide a "help" option that is displayed either upon request or when incorrect arguments are supplied. Let us see how to accomplish this task.

First, we must explain how terminal arguments work. Terminal arguments are specified after the executable (name) and are separated by spaces. For instance, if our program name is `calculator`, we might use `./calculator --add 5 4 17`. Thus, `args[0]` is `--add`, `args[1]` is `"5"`, `args[2]` is `"4"`, and `args[3]` is `"17"`. For simplification purposes, we will assume that the first argument is always the operation/help flag, and the remaining values are operands. This means that the program should output `26`. Let us write a method that parses the operation/help flag. Upon success, it returns `true` and upon failure, it returns `false`. This prevents the program from further interpreting bad terminal arguments, e.g., `./calculator --wrong 5 12`. We will also use `false` as an indication that the `help` menu was requested or prompted. Thus, to not duplicate code, we should write another method that displays the relevant program usage information.

Listing 3.12—Writing a Calculator and Parsing Terminal Arguments

```
class Calculator {

  public static void main(String[] args) {
    if (parseCommand(argv[0])) {
      // Continue.
    }
    // Otherwise, stop.
  }

  static boolean parseCommand(String cmd) {
    if (cmd.equals("--add") || cmd.equals("--sub")) {
      return true;
    } else {
      displayHelp();
      return false;
    }
  }
```

```
  static void displayHelp() {
    System.out.println("usage: ./calculator --(help | add | sub) <n1> [n...]");
  }
}
```

Up next is the process of interpreting each valid operation, i.e., `--add` and `--sub`. The former will add each successive argument one-by-one while the latter subtracts them from left-to-right. Of course, because we receive the terminal arguments as strings, we will need to convert their values from strings to double values using `Double.parseDouble`. For the time being, we will assume that these *are*, in fact, double values, rather than working through the painstaking process of parsing a string for the existence of a proper `double` datatype value. We encourage the readers to implement this method themselves, along with the appropriate tests.

Note that in the code below, we utilize an `if/else if` combination without an accompanying `else`, which we would normally discourage. Because we exhaust the possibilities with `parseCommand`, however, we will allow its usage. The `parseAdd` and `parseSub` methods are trivial and we have shown an example of their implementation previously, so we will also omit these to preserve space and avoid unnecessary repetition.

Listing 3.13—Main Method for Calculator

```
class Calculator {

  public static void main(String[] args) {
    if (parseCommand(args[0])) {
      String cmd = args[0];
      double[] operands = convertToDoubleArray(args);
      if (cmd.equals("--add")) {
        System.out.println(parseAdd(operands));
      } else if (cmd.equals("--sub")) {
        System.out.println(parseSub(operands));
      }
    }
  }
}
```

Example 3.6. Let's write a program that receives a list of integers through the terminal, as an "argument array" of sorts, and allow the user to pass flags to denote the operation to perform on the list. Our program will support the following operations: `--sum`, `--product`, `--min`, and `--max` command. We will also provide a `--help` flag that displays the program usage information. This is a substantial project, but doing so allows us to practice using arrays and integrating more complex terminal arguments. As a measure of simplification, we will assume that the first n arguments are the numeric values, and the remaining arguments are the operation flags. Let's further assume that the user will not pass an invalid command. Lastly, we shall not consider any non-sensical in a given context, e.g., the minimum/maximum of no input values. The first terminal argument denotes the number of values to expect, so we will use this to initialize our array.

To start, let's see a few example runs of our program, containing a mixture of flags.

```
./ArrayArguments 5 1 2 3 4 5 --sum --max
sum: 15.000000
max: 5.000000
./ArrayArguments 3 100 200 -100 --product --sum --min
sum: 200.000000
```

```
product: -2000000.000000
min: -100.000000
./ArrayArguments --help
usage: ./ArrayArguments <n> <n1> [n...] [--(sum | product | min | max)]
```

The ordering of the output is irrelevant, and depends on how we parse the input flags in the main method. To scan for a given flag, let's write a static method to return whether or not the flag exists in the arguments array.

Listing 3.14—Scanning for a Flag Tester

```java
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static ArrayArguments.isFlagPresent;

class ArrayArgumentsTester {

  @Test
  void testScanForFlag() {
    String[] args = new String[]{"--sum", "--max", "--min"};
    assertAll(
      () -> assertEquals(true, isFlagPresent(args, "--sum")),
      () -> assertEquals(true, isFlagPresent(args, "--max")),
      () -> assertEquals(true, isFlagPresent(args, "--min")),
      () -> assertEquals(false, isFlagPresent(args, "--product")));
  }
}
```

Listing 3.15—Scanning for a Flag Implementation

```java
class ArrayArguments {

  /**
   * Returns whether or not a given flag exists in the arguments array.
   * @param args array of arguments.
   * @param flag flag to search for.
   * @return whether or not the flag exists.
   */
  static boolean isFlagPresent(String[] args, String flag) {
    for (String arg : args) {
      if (arg.equals(flag)) { return true; }
    }
    return false;
  }
}
```

The other "operations" methods, as well as their tests, are simple to write, and will omit their implementation.

Our main method first checks to see if the user entered the "help" command and, if so, presents the necessary information for runnning the program. Otherwise, we perform a case analysis on the terminal arguments, looking for the presence of the operation flags. For arbitrary reasons, we output the sum, then the product, then the min, and finally the max, in that order, despite the ordering of the flags. As an exercise, we encourage the readers to modify the program to output the values in the order of the flags. An important detail that some may miss is that we use a sequence of `if` statements rather than `if/else if` statements. This is because we want to allow the user to pass multiple flags, and we do not want to restrict them to only one. Thus, we must check for the presence of each flag individually.

Listing 3.16—Main Method for Array Arguments

```java
class ArrayArguments {

  public static void main(String[] args) {
    if (isFlagPresent(args, "--help")) {
      displayHelp();
    } else {
      int n = Integer.parseInt(args[0]);
      double[] values = new double[n];
      for (int i = 0; i < n; i++) { values[i] = Double.parseDouble(args[i + 1]); }
      if (isFlagPresent(args, "--sum")) {
        System.out.printf("sum: %f\n", sum(values));
      }
      if (isFlagPresent(args, "--product")) {
        System.out.printf("product: %f\n", product(values));
      }
      if (isFlagPresent(args, "--min")) {
        System.out.printf("min: %f\n", min(values));
      }
      if (isFlagPresent(args, "--max")) {
        System.out.printf("max: %f\n", max(values));
      }
    }
  }
}
```

Example 3.7. Arrays can be of arbitrary dimension and are not restricted to only one. In this problem we will make use of a two-dimensional array, which might be thought of as a matrix or a grid. We will write a method that returns the sum of the elements of a two-dimensional array of integers. Traversing over an n-dimensional array generally involves nested loops. The order in which we traverse over the array can be significant. For example, the following code uses *row-major* ordering, since we iterate over the rows first, and then the columns, meaning that we visit the elements in the order $A_{0,0}, A_{0,1}, A_{0,2}, \ldots, A_{1,0}, A_{1,1}, A_{1,2}, A_{2,0}, A_{2,1}, A_{2,2}$. Conversely, *column-major* ordering would visit the elements in the order $A_{0,0}, A_{1,0}, A_{2,0}, \ldots, A_{0,1}, A_{1,1}, A_{2,1}, A_{0,2}, A_{1,2}, A_{2,2}$.

Multi-dimensional arrays are nothing more than arrays of arrays. Thus, as an example, we can declare a 3×4 two-dimensional array of integers (with three rows and four columns) as follows:

```java
int[][] A = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
```

Listing 3.17—Tests for the Sum of a Two-Dimensional Array

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

class SumOf2DArrayTester {

  @Test
  void sumOf2DArrayTest() {
    int[][] A = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
    assertAll(
```

```
      () -> assertEquals(78, sumOf2DArray(A)),
      () -> assertEquals(0, sumOf2DArray(new int[][]{})),
      () -> assertEquals(1, sumOf2DArray(new int[][]{{1}})));
  }
}
```

We need to know both the number of rows and the number of columns to traverse over a two-dimensional array. To retrieve the number of rows, we simply refer to the array's length via A`.length`. To get the number of columns, again, because we know that A is an array of one-dimensional arrays, we use $A[0]$`.length`, or in general, $A[i]$`.length` for any i such that $0 \leq i < A$`.length`.[1]

Listing 3.18—Sum of a Two-Dimensional Array

```
class SumOf2DArray {

  /**
   * Computes the sum of the values in a two-dimensional array.
   * @param arr two-dimensional array of integers.
   * @return sum of those values.
   */
  static int sumOf2DArray(int[][] arr) {
    int sum = 0;
    for (int i = 0; i < arr.length; i++) {
      for (int j = 0; j < arr[i].length; j++) {
        sum += arr[i][j];
      }
    }
    return sum;
  }
}
```

Note that to access the element at row i and column j, we use $A[i][j]$.

Example 3.8. Let's solve a slightly harder problem using two-dimensional arrays. Suppose that we want to write a method that returns the number of possible moves that a rook can take to go from the top-left of a (not-necessarily rectangular) board to the bottom-right, assuming that it cannot move left or up. The naive solution to this problem is to use a recursive method that changes its position by one in either direction, stopping once we hit the bottom-right of the board. Assuming the rook starts at (x, y) and the board is $n \times m$, we can write the following method.

Listing 3.19—Tests for the Rook Problem

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

class RookPathTester {

  @Test
  void testRookPath() {
    assertAll(
      () -> assertEquals(1, rook(0, 0, 1, 1)),
      () -> assertEquals(2, rook(0, 0, 2, 2)),
      () -> assertEquals(6, rook(0, 0, 3, 3)),
      () -> assertEquals(10, rook(0, 0, 4, 4)));
```

[1] This generalization applies because arrays in Java cannot be *ragged*, or where different rows/columns have different lengths.

```
  }
}
```

Listing 3.20—Rook Problem Implementation

```
class RookPath {

  /**
   * Computes the number of possible paths that a rook can take to go from the
   * top-left of a board to the bottom-right, assuming that it cannot move left
   * or up.
   * @param x x-coordinate of the rook's starting position.
   * @param y y-coordinate of the rook's starting position.
   * @param n number of rows of the board.
   * @param m number of columns of the board.
   * @return number of possible paths.
   */
  static int rook(int x, int y, int n, int m) {
    if (x == n || y == m) { return 1; }
    else { return rook(x + 1, y, n, m) + rook(x, y + 1, n, m); }
  }
}
```

Much like how the recursive definition of Fibonacci is horrendously slow, so is this implementation. We need something faster, and indeed, we can take advantage of a two-dimensional array because of an emerging pattern. Notice that in the bottom-right corner, there is only one possible solution. We can generalize this to say that there is only one solution for any position in the bottom row or the far-right column. From here, we can work our way up and to the left, filling in the number of possible solutions for each position.

For example, the position $(n-1, m-1)$ has a value of two, since it can move either right or down. The position $(n-2, m-1)$ has a value of three, since it can move right, down, or down and then right. We can continue this process until we reach the top-left corner, which will have the value of the number of possible paths from $(0, 0)$ to (n, m).[1] We can write a method that computes this value using a two-dimensional array. Composing the solution in this manner is called *dynamic programming*, which comes up often when attempting to optimize problems that have naive and outrageously recursive solutions.

To prevent our code from going out of bounds, we need to add one to the bounds of our input array. That is, if we want to compute the number of possible paths from $(0, 0)$ to (n, m), we need to create an array of size $(n+1) \times (m+1)$, because the current value of the array at (n, m) depends on the values of the array at $(n+1, m)$ and $(n, m+1)$.[2]

Dynamic programming problems are often solved using two-dimensional arrays using the following three-step process:

1. For a problem size of n and m, initialize a two-dimensional array of size $(n+1) \times (m+1)$.
2. Populate the array with the necessary base cases.
3. Iterate over the array, filling in the values using a recurrence relation.

[1] Should we want to choose an arbitrary starting point, we can retrieve that index rather than $(0, 0)$ in the resulting two-dimensional array.

[2] When writing dynamic programming algorithms, it is commonplace to call the array `dp` out of convention.

Listing 3.21—Tests for the Rook DP Problem

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

class RookPathTester {

  @Test
  void testRookPathDP() {
    assertAll(
      () -> assertEquals(1, RookPath.rook(2, 3, 2, 3)),
      () -> assertEquals(2, RookPath.rook(1, 2, 2, 3)),
      () -> assertEquals(6, RookPath.rook(0, 1, 2, 3)),
      () -> assertEquals(10, RookPath.rook(0, 0, 2, 3)));
  }
}
```

Listing 3.22—Rook DP Problem Implementation

```
class RookPath {

  /**
   * Computes the number of possible paths that a rook can take to go from the
   * (x, y) position of a board to the bottom-right, assuming that it cannot
   * move left or up. This approach uses dynamic programming.
   * @param x x-coordinate of the rook's starting position.
   * @param y y-coordinate of the rook's starting position.
   * @param n number of rows of the board.
   * @param m number of columns of the board.
   * @return number of possible paths.
   */
  static int rook(int x, int y, int n, int m) {
    int[][] dp = new int[n + 1][m + 1];

    // Compose the initial bottom-row solutions.
    for (int i = 0; i < n + 1; i++) { dp[i][m] = 1; }

    // Compose the initial far-right solutions.
    for (int i = 0; i < m + 1; i++) { dp[n][i] = 1; }

    // Now do the dynamic programming algorithm.
    for (int i = n - 1; i >= 0; i--) {
      for (int j = m - 1; j >= 0; j--) {
        dp[i][j] = dp[i + 1][j] + dp[i][j + 1];
      }
    }
    return dp[x][y];
  }
}
```

3.2 Collections

In this section we will introduce the *Java Collections API*. In doing so we will discuss three broad classifications of data structures provided by the API:

1. Sequential-based
2. Dictionary-based

3. Set-based

Note that our discussion is not all-inclusive of every data structure in the API, but we present those that we feel are most valuable to this course.

Sequential-Based Data Structures

We categorize data structures that have an ordering over the natural numbers as *sequential-based.* That is, each element has an index where it "lives" for its lifetime. Each index is, similar to standard arrays, numbered from zero to the size of the collection minus one. Let us now take a dive into these different collections.

`ArrayList` Class

Arrays are fixed-size data structures; once they are initialized, they cannot, themselves, be resized. A solution to this problem is to create a new array A' of the same type with a new size and copy the elements from the old array to A'. Doing so is not difficult but cumbersome to repeatedly implement. Consider a situation in which the number of elements to store is unknown at compile-time. We, therefore, cannot use an array without repeated resizing. The correct and colloquial solution involves the `ArrayList` class.

First, however, let us see how we might go about implementing a *dynamic array*, called a *list.* using only methods. Suppose we want to store positive integers in this list. We also want to be able to add, set, and retrieve elements at a specified index. We will continue to work with arrays for the time being to demonstrate what goes wrong with this ideology, and then to understand the power of the `ArrayList`.

We need a few methods to solve this problem: `makeList`, `addToList`, `getFromList`, and `setInList`. At the end of the day, we want the programmer who uses these methods to not worry about resizing the array themselves; the logic within handles the dirty work.

To better relate to the `ArrayList` class implementation, we will write two versions of the `makeList` method: one that receives an initial size and one that does not. Designing two methods of the same name that receive different parameter types/quantities is known as *method overloading*, and we will see this further in our discussion on *classes.* `makeList` returns an array of integers instantiated to the given size, or a base size of ten elements in the method that does not receive a parameter. Note that, inside of the `makeList` method that does not receive a parameter, we invoke `makeList(10)` so as to not repeat ourselves.

Listing 3.23—Dynamic Integer Array Method "Constructors"

```
class DIntArray {

  /**
   * Creates an array of the given size.
   */
  static int[] makeList(int size) {
    int[] array = new int[size];
    for (int i = 0; i < array.length; i++) {
      array[i] = -1;
    }
    return array;
  }

  /**
```

```
   * Creates an array with ten spaces.
   */
  static int[] makeList() {
    return makeList(10);
  }
}
```

We now want a method that will add a value to a given "list" in this fashion. In particular, we know that indices whose elements are -1 correspond to "free/available" slots for the next value to-be added. The thing is, there is more to consider than just replacing the first-found instance of -1 with the desired value. We need to ensure that room exists for this new value; i.e., whether or not there is a -1 to begin with. As such, we should write a local helper method that returns a resized list with the values copied over from the old list; the only difference being a doubling in element capacity. Then, inside `addToList`, we check to see if we were able to properly insert v into the list and, if not, we resize and make a recursive call to `addToList`.[1] Regarding performance and memory usage, this is not an optimal solution since we could simply add v to index $|A|$ of the new array A', since we know $|A'| = |A|$.

Listing 3.24—Adding Value into Dynamic Array Tests

```
import static Assertions.assertAll;
import static Assertions.assertArrayEquals;
import static DIntArray.makeList;
import static DIntArray.add;

class DIntArrayTester {

  @Test
  void addTest() {
    int[] arr1 = makeList(5);
    int[] arr2 = addToList(arr1, 20);
    int[] arr3 = addToList(arr2, 350);
    assertArrayEquals(new int[]{20, -1, -1, -1, -1}, arr2);
    assertArrayEquals(new int[]{20, 350, -1, -1, -1}, arr3);
  }
}
```

Listing 3.25—Resize Array Method Implementation

```
class DIntArray {

  /**
   * Doubles the capacity of a list, returning a
   * new list with the old elements copied over.
   */
  private static int[] resize(int[] list) {
    int[] newList = new int[list.length * 2];
    for (int i = 0; i < list.length; i++) {
      newList[i] = list[i];
    }
    return newList;
  }
}
```

[1] We could use `Arrays.copyOf`, but it is important to understand *how* the copying occurs.

Listing 3.26—Adding Value into Dynamic Array Implementation

```
class DIntArray {

  /**
   * Adds a value to the next-available spot in the list.
   * We define next-available as the first -1 we find from the left.
   */
  static int[] addToList(int[] list, int v) {
    boolean added = false;
    int[] newList = makeList(list.length);
    for (int i = 0; i < list.length; i++) {
      // If we haven't inserted the value yet and we found
      // a free slot, insert it and mark added as true.
      if (list[i] == -1 && !added) {
        newList[i] = v;
        added = true;
      } else {
        // Otherwise, just copy over the old value.
        newList[i] = list[i];
      }
    }
    if (!added) { return addToList(resize(newList), v); }
    else { return newList; }
  }
}
```

We have two methods to go: `getFromList` and `setInList`. The former retrieves an element at a given index and the latter replaces the element at a given index. Both methods receive an index i that must be in-bounds, where in-bounds refers to not only the bounds of the array, i.e., neither negative nor exceeding the length of the list, but also the logical indices. The *logical indices* of a list are the indices in which elements exist. For our purposes, these indices are from zero up until and excluding the first instance of -1. We will also write a helper method that retrieves the index of the first "free" slot of a list, i.e., the first occurrence of -1.

Listing 3.27—"Get From List" Dynamic Array Implementation

```
class DIntArray {

  static int getFromList(int[] list, int idx) {
    int upperBound = getFirstFreeSlot(list);
    if (idx < 0 || idx >= upperBound) {
      return -1;
    } else {
      return list[idx];
    }
  }

  private static int getFirstFreeSlot(int[] list) {
    for (int i = 0; i < list.length; i++) {
      if (list[i] == -1) { return i; }
    }
    return -1;
  }
}
```

Listing 3.28—"Set In List" Dynamic Array Implementation

```
class DIntArray {

  static int[] setInList(int[] list, int idx, int v) {
    int upperBound = getFirstFreeSlot(list);
    if (idx < 0 || idx >= upperBound) {
      return -1;
    } else {
      return list[idx];
    }

    // Copy over old elements.
    int[] newList = makeList(list.length);
    for (int i = 0; i < list.length; i++) {
      if (i == idx) {
        newList[i] = v;
      } else {
        newList[i] = list[i];
      }
    }
    return newList;
  }
}
```

It should be noted that our implementation is a *functional list*, which means that the (old) passed list, in and of itself, is not altered. Rather, we create a new list with each successive modification.

So, the problems of relying only on methods become clear: we have no way of keeping track of when/where the last *logical element* is located. By assuming an input of only positive integers, we can say that the first occurrence of a -1 marks the next-available spot to add a value. Sentinel indicators like these fall apart once we allow for different inputs, e.g., negative numbers.

The Java `ArrayList` class is a dynamic list data structure, is our first look at a "powerful" data structure insofar as its capabilities are concerned. Additionally, it is the first class that incorporates *parameterized types*. With arrays, we must specify the type upon declaration; the `ArrayList` class is *generic* in the sense that it operates on any type, whether that is `Integer`, `String`, or anything else, making it an incredibly flexible data structure.

What makes an `ArrayList` so convenient is the abstraction and encapsulation of the underlying data structure. Underneath these lies a primitive array that is resized whenever necessary, similar to our resizing method. Thankfully, us as the programmers need not to worry about its implementation, but understanding it is key to grasping just what makes it better than our previous approach of writing static methods that receive and return lists. First, like we said, our methods-based implementation of dynamic lists is restricted to one datatype, namely `int`, and also uses -1 as the "sentinel." Conversely, `ArrayList` stores a number that references the next-available spot, meaning we do not need to waste time traversing the list for each and every instance of adding or modifying elements.

To declare an `ArrayList` called A, we write the following, where T is a class representing the type of elements contained within:

```
List<T> A = new ArrayList<>();
```

Java Array Lists

An *ArrayList* is a dynamically-sized data structure for storing elements.

List<T**>** A = **new ArrayList<>()** creates an `ArrayList` of type T named A.

A**.get(**i**)** retrieves the element at index i^{th} of A. We refer to this as position $i + 1$.

A**.set(**i, v**)** assigns v to index i of A.

A**.size()** returns the number of logical elements in the list, i.e., the logical size.

A_1**.equals** (A_2) returns whether or not the elements of A_1 are equal to the elements of A_2, using the `.equals` method implementation of A_1.

A**.toString()** returns a string representation of the elements in A, separated by commas and enclosed by brackets.

Collections.sort (A) performs an in-place sort of A, meaning the contents of A are modified.

Figure 3.2: Useful `ArrayList`-based Methods.

This initializes a `List` A, but instantiates it as a new `ArrayList`.[1] Notice that we do not specify the number of elements to store like we would an array. Indeed, because lists are dynamic in size, there is no need to specify a default. Java defaults the starting size of a newly-declared `ArrayList` to ten elements, although this can be changed by passing a size argument between the parentheses.

```
List<T> A = new ArrayList<>(100);
```

You might be tempted to ask, "Why might that be necessary?", which is a great question. Remember that resizing an array depends on the number of pre-existing elements. Thus, the fewer resizes there are, the better, hence why we double the array capacity in our functional list implementation. If we know that we might have a lot of elements to add to the list from the start, it is a good idea to specify this as a parameter to the `ArrayList`. On average, adding a value to the end of an `ArrayList` is a *constant cost*, or occurs instantaneously, because we know exactly where the next free spot is located. We cannot forget the time it takes to resize, however, so we declare that the `.add` method takes constant time with respect to *amortized analysis*. In essence, sometimes we have to perform a resize-and-copy operation, but on average, we do not need to consider the cost thereof.

Like arrays, modifying elements in-place and retrieval also has a constant cost; the underlying data structure is an array after all. We retrieve a value at a given index using `.get`, and we replace an existing value at a given index using `.set`.

Example 3.9. If we instantiate an `ArrayList` of integers `ls1`, we can perform several operations to demonstrate our understanding.

```
List<Integer> ls1 = new ArrayList<>();
ls1.add(439);
ls1.add(311);
ls1.add(654);
ls1.add(523);
```

[1] The reason behind this *polymorphic* choice will become apparent in subsequent chapters.

```
ls1.toString();  => {439, 311, 654, 523}
ls1.get(0);      => 439
ls1.set(1, 212);
ls1.add(677);
ls1.toString()   => {439, 212, 654, 523, 677}
```

Finally, we can remove an element, when given its index, using `.remove`. Be aware that removing elements is not as simple as adding. Suppose that, from the previous example, we invoke `ls1.remove`(2), which removes the value 654. We cannot simply have a slot/index with a missing value. So, Java compensates by shifting all values to the right of the removed value over by one.

Example 3.10. Consider a situation in which we have an `ArrayList` that contains $1,000,000$ arbitrary integers. If we continuously remove elements from the front of the list, we shift each and every value down by one index. Propagating this through all one million elements results in a hefty cost of $999,999 + 999,998 + \cdots + 3 + 2 + 1$ shifts. So, removing n elements from the front of an `ArrayList` is representable as an equation of time T.

$$T(n) = (n-1) + (n-2) + (n-3) + \cdots + 3 + 2 + 1$$

We can collapse this into an arithmetic series from 1 to $n - 1$:

$$\begin{aligned} T(n) &= \sum_{i=1}^{n-1} i \\ &= n \cdot \left(\frac{1+n-1}{2}\right) \\ &= n \cdot \left(\frac{n}{2}\right) \\ &= \frac{n^2}{2} \end{aligned}$$

Therefore, to remove $1,000,000$ elements from the front of an `ArrayList`, we need to perform roughly $1,000,000^2/2$ operations, which is an astronomical number, even for computers of today. Removing elements from other indices, excluding the rear, will incur similar, yet smaller, shifting cost penalties. As we will see later in this chapter, other data structures are significantly better choices if it is a desire to quickly poll elements from the front.

Example 3.11. We made a big deal about growing the underlying array when we run out of room to add new elements. We could ask a similar question about what to do when we, say, remove a ton of elements from a large list. If our list contains one million elements and we then clear the list, it makes little sense, at first glance, to have space allocated for one million non-existent values. Though, Java's implementation of `ArrayList` does not decrease the size of the backing array, preferring performance over memory usage. The reasoning behind this choice is straightforward: suppose we have an `ArrayList` of 500 elements and we remove 250 elements. Is it fair to decrease the size of the list by a factor of two? If so, what happens when we add one more element, totaling to 251? We then have to grow the list, again, wasting valuable time.

Example 3.12. Consider the following code. What does s contain after execution?

```
List<Integer> ls1 = new ArrayList<>();
ls1.add(10);
List<Integer> ls2 = ls1;
ls2.add(20);
ls1.add(30);
String s = ls1.toString() + " " + ls2.toString();
```

Should you be unaware of aliasing, you might say it resolves to `"[10, 30] [20]"`. *Aliasing* is a form of object-sharing. When allocating any type of *object*, whether that object is an array, an `ArrayList`, a `String`, or something else, we assign a reference *to* that object in memory via the variable declaration. That is, in the preceding code, `ls1` references the location, in memory, of an `ArrayList`. Correspondingly, when we declare `ls2` and assign to it `ls1`, we are not copying over the values from `ls1` into `ls2`; we are expressing that `ls2` should reference the same list as referenced by `ls1`. Therefore by asserting `ls2` as an alias of `ls1`, any modifications made to either is reflected when referencing the other. In this instance, s resolves to `"[10, 20, 30] [10, 20, 30]"`

Example 3.13. Consider two methods `void increment(int x)`, which increments an integer variable, and `void increment(ArrayList<Integer> ls, int idx)`, which increments the value at a given index in a list of integers.

```
static void incrementInt(int x) {
  x = x + 1;
}
static void incrementList(ArrayList<Integer> ls, int idx) {
  ls.set(idx, ls.get(idx) + 1);
}
```

If we then pass a variable to the `incrementInt` method, we might expect the resulting value, outside of the method, to also be incremented. Unfortunately, that is not what happens–in the following code segment, we see that the primitive variable `y` remains unchanged outside of the method invocation. This happens because primitive values are *passed by value*. In essence, methods receive a copy of the value, which means that the original variable is not modified, and the change made inside the scope of `incrementInt` is rendered useless. Compare this to what happens if we pass an `ArrayList<Integer>` to `incrementList`: we see that the change occurs both inside and outside the scope of the method body. Objects, e.g., `String`, `ArrayList`, arrays, and so forth, when supplied as arguments to methods, are passed by a paradigm that we will call *pseudo-reference*.[1] Passing by pseudo-reference means to suggest that we are not truly passing the argument by reference, and this is correct. Objects in Java are still passed by value, but instead of creating a copy of the object, the object's corresponding hashcode is supplied to the method, which is roughly equivalent to a memory reference.[2] Therefore, any changes made to the value inside the method are reflected outside.

```
int y = 5;
assertEquals(5, y);          // Assertion before increment.
incrementInt(y);
assertEquals(5, y);          // Assertion after increment.
```

```
List<Integer> ls = new ArrayList<>();
```

[1] The emphasis on calling this "pseudo-reference" is to provide an intuition for those readers who may know about true pass-by-reference, while satisfying those who want to angrily shout that Java is strictly pass-by-value.

[2] In Chapter 4, we will discuss object hashcodes in greater detail.

```
ls.add(5);
assertEquals(5, ls.get(0)); // Assertion before increment.
incrementList(ls, 0);
assertEquals(6, ls.get(0)); // Assertion after increment.
```

Example 3.14. We have seen the repeated use of the `Integer` and `Double` classes when parameterizing the types for `ArrayList`, but what is the point? Could we not instead opt for `int` and `double`, as we have traditionally? The answer is a resounding no; we must take advantage of the luxury that is *autoboxing* and *autounboxing* through the *wrapper classes*. The classes `Integer`, `Double`, as well as `Short`, `Byte`, `Long`, `Char`, and `Boolean` are classes that encapsulate, or box, a corresponding primitive value. Parameterized types only work with class types; primitive datatypes are disallowed. So, for instance, if we want to work with an `ArrayList` of `int` elements, we are required to use the `Integer` wrapper class in our type declaration. We can, of course, declare an `Integer` using the following overly-verbose syntax:

```
Integer x = new Integer(42);
int y = x.getValue(); // y = 42.
```

We explicitly wrap the primitive integer 42 in x, which is of the `Integer` type, then unwrap its value to be stored in y. Manually wrapping and unwrapping primitives is tiresome and only introduces redundant code, hence why Java autoboxes and autounboxes as needed. For example, if we declare an `ArrayList` of `Integer` values, then add the primitive integer literal `42` to the list, Java will autobox the literal into its `Integer` wrapper. Going the other direction, if we want to iterate over the values in the list, we might use the enhanced-for loop. If Java did not support autounboxing, we would instead have to use `Integer` as opposed to `int` in the loop variable declaration.

```
ArrayList<Integer> al = new ArrayList<>();
al.add(42);
for (int e : al) {
  ...
}
```

Example 3.15. Testing is always important when writing programs, as this text has emphasized from the first page. Writing test cases for lists, naively, is cumbersome due to the repetition of `.add` method calls when populating the list. Instead, Java has the convenient `List.of` method, which receives any number of arguments. Let us see an example of this method.

```
// Old way:
List<Integer> ls1 = new ArrayList<>();
ls1.add(5);
ls1.add(40);
ls1.add(4);
ls1.add(42);
// New way:
List<Integer> ls2 = new ArrayList<>(List.of(5, 40, 4, 42));
```

Using `List.of` raises a question: does `List.of` only receive five integer arguments? What if I want to specify more than five, or less than five? The answer to this excellent question is that `List.of` is a *variadic-argument method*. Methods may be written to receive any number of arguments, which are then collapsed into a data structure that is traversable.

Example 3.16. Suppose we want to write a variadic method that computes the average of any number of given `double` values. Without variadic-argument methods, we must wrap these in a list or an array, then pass it to the method. Variadic arguments allow us to specify any desired number of arguments, and it is the responsibility of the method to interpret those values as a traversable data structure. So, to iterate over such a structure, we can use the enhanced-for loop.

Listing 3.29—Variadic Argument `numAverage` *Tester*

```java
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static NumAverage.numAverage;

class NumAverageTester {

  @Test
  void numAverageTest() {
    assertAll(
      () -> assertEquals(0, numAverage()),
      () -> assertEquals(5.66, numAverage(5, 2, 10)),
      () -> assertEquals(55.2, numAverage(10, 65, 77, 81, 43)));
  }
}
```

Listing 3.30—Variadic Argument `numAverage` *Implementation*

```java
class NumAverage {

  /**
   * Computes the average of a sequence of provided integers.
   * @param variadic integers.
   * @return average if there is at least one number or zero otherwise.
   */
  static int numAverage(int ... nums) {
    double sum = 0;
    for (int e : nums) { sum += e; }
    return nums.length == 0 ? 0 : sum / nums.length;
  }
}
```

Example 3.17. If we want to, say, specify that a method must receive at least two parameters, those being a `String` and an `int`, followed by zero or more `String` values, we may declare the first two parameters, then incorporate the variadic notation for the last. Doing so ensures that we pass the required parameters, but any thereafter are optional but variadic nonetheless.

Listing 3.31—Requiring Parameters with Variadic Arguments

```java
class RequiredVariadicParameters {

  static int doSomething(String s, int v, String ... vals) { ... }
}
```

Java Linked Lists

A *LinkedList* is a node-based data structure where each element contains a link to its successor (and potentially predecessor).

`List<`T**`>`** $A =$ **`new LinkedList<>()`** creates a `LinkedList` of type T named A.

A**`.get(`**i**`)`** retrieves the element at index i^{th} of A. We refer to this as position $i + 1$.

A**`.set(`**i, v**`)`** assigns v to index i of A.

A**`.size()`** returns the number of logical elements in the list, i.e., the logical size.

Figure 3.3: Useful `LinkedList`-based Methods.

`LinkedList` Class

Linked lists remove us from the shackles of array-based data structures, in that, as their name implies, they are a series of *nodes*, or elements, linked together in a chain of sorts. These nodes need not be adjacent in memory, but rather reference each other to find what comes next in the chain/list. For instance, if we create a linked list, it has a *front*/*head* element that always references, or points, to the first element in the list (upon initialization, the head refers to nothing). If we add a new element, the head now points to this first element. Subsequent additions to the list continue growing the chain and links. Namely, element 1 points to element 2, element 2 to 3, and so on.

Elements have an associated index and value, but linked lists are not constrained to a static size even in the underlying implementation! So, we can add and remove links from the chain whenever we please with no shuffling of values around aside from links within the chain. Removing elements from the front of a linked list is constant time, and scales linearly with the number of elements in the list rather than a quadratic growth.

Of course, these advantages are not without their disadvantages. Reading and modifying elements are slower operations than the array counterparts since the elements are not contiguous blocks in memory. Recall that with an `ArrayList`, because elements are placed side-by-side in memory, we know the location of any arbitrary index in the array list by a multiplicative offset of the starting index and the "byte-size" of each element. Linked lists do not provide this guarantee and should not be treated as such. Adding and removing elements are "faster" in the sense that, as we stated, copying values over to a new array is out of the question. Because of this, though, we need to iterate/traverse through the list each time we wish to reference a provided index. The same goes for inserting elements into the list. Adding or removing elements from the front or rear of the list, on the other hand, are instant operations since we keep track of the first element of the list (and we can, similarly, keep track of the last!). Linked lists are also the backbone of many other data structures as we will soon see.

`Stack` Class

Imagine you are washing dishes, by hand, at the kitchen sink. These dishes are assorted in a single stack to your left. A dish cannot be removed from anywhere but the top of the stack because displacement anywhere else will destroy the stack. Additionally, further

Java Stacks

A *Stack* is a last-in-first-out (LIFO) data structure where each element is linked to the element immediately below.

`Stack<T> S = new Stack<>()` creates a `Stack` of type T, named S.

S`.peek()` returns, but does not remove, the top element of S.

S`.pop()` returns and removes the top element of S.

S`.push(`e`)` pushes e to the top of S, making e the element on the top of the stack.

S`.size()` returns the number of logical elements in the stack.

Figure 3.4: Useful `Stack`-based Methods.

imagine that people are, to your dismay, adding more dishes to the stack. Again, dishes cannot be added anywhere else but the top of the stack.

The *stack* data structure is as simple as it sounds—a collection of elements that operate on the principle of last-in-first-out, or LIFO. In other words, the last thing that we enter is the first thing removed. Stack implementations contain at least the following operations: POP and PUSH, where the former removes the top-most element from the stack (if one exists), and the latter adds a new element to the top of the stack. There may also exist an operation to view, but not remove, the top-most element via PEEK.

Stacks have the advantages of instant insertion and removal times but are obviously not as flexible as an array or linked list. A practical example of a stack data structure would be an "undo" function in a document-editing program—whenever an action is made, it is pushed to an event stack. An "undo" event would resemble popping an action off this stack. We illustrate this concept in Figure 3.5.

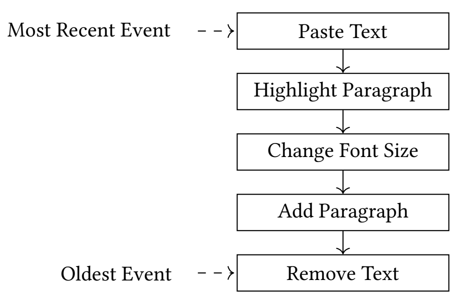


Figure 3.5: Example of "Undo" Event Stack in Text-Editing Program

`Queue` Interface

Imagine you are in line at an amusement park for the most intense roller coaster in the world. Another, perhaps more generic term for a "line" is a *queue*. In this metaphor, riders enqueue the line at the back and board the roller coaster (and hence dequeue from the line) at the front.

What we have described is a practical example of the queue data structure. In a queue, elements are enqueued, or inserted, to the back of the line and are dequeued, or removed, from the front. Queues operate on the principle of first-in-first-out, or FIFO. The implementation of a queue data structure may contain different names for their operations, but at their core should contain operations for inserting an element to the back of the queue (e.g., ENQUEUE) and removing an element from the front of the queue (e.g., DEQUEUE).

Like the operations of a stack, these are also constant-time, since we keep a reference to the front and rear elements of a queue. Queues, consequently, share similar drawbacks to stacks in that elements are not randomly accessible, i.e., we only know what exists at the front and rear of a queue instantaneously. Figure 3.6 demonstrates the task queue of a printer, which has a sequence of files to print one after the other.

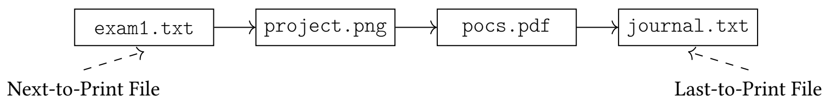


Figure 3.6: Example of Printer Task Queue

Unfortunately and inconveniently, there is no `Queue` class in Java. Instead, `Queue` is an *interface* that other classes implement whose structure resembles a queue. To create a first-in-first-out queue data structure, we may initialize a variable to be a `Queue` then instantiate it as a `LinkedList`. Thankfully, the `LinkedList` class contains all relevant methods for operating a FIFO-based queue.

```
Queue<Integer> q = new LinkedList<>();
```

Treating `q` as a queue rather than a linked list is easy thanks to the methods supplied by the `LinkedList` implementation. Figure 3.7 shows some of these handy methods.

`PriorityQueue` Class

Priority queues are the final sequential data structure that we will discuss from the Collections API. Though, placing it in this section is slightly disingenuous because, while priority queues have an ordering in their underlying data structure, saying that elements correspond to an index is largely incorrect. Priority queues, as their name suggests, rank items in the queue by a score called the *priority*. Elements with the highest priority are at the "front" of the priority queue. Inserting elements into a priority queue potentially alters the positioning of preexisting elements.

Priority queues base priority on one of two contributing factors: either the *natural ordering* of elements, or a *comparator object*. The natural ordering of elements is straightforward: natural ordering for numbers is their standard numeric ordering. For strings, the natural ordering is by lexicographical ordering. These, however, are not as interesting as comparators, which we will now discuss.

A *comparator* is a way of comparing two arbitrary "things," whether these things are numbers (i.e., the wrapper classes), strings, or another kind of object, we can define custom ways of comparing *any* non-primitive datatype.

Example 3.18. Let us design a `Comparator` for prioritizing strings that start with the lowercase letter `'p'`. Comparators are constructed like other objects via `new`, but something interesting about their implementation is that we must specify *how* to compare two

Java Queue

A *Queue* is a first-in-first-out (FIFO) sequential data structure where each element is linked to the element immediately after.

`Queue<`T**`>`** Q = **`new LinkedList<>()`** creates a `Queue` of type T, named Q.

Q**`.addLast`**(e) adds e to the end of Q, placing it at the end of the queue structure.

Q**`.poll`**$()$ returns and removes the element from the front of the queue.

Q**`.peek`**$()$ returns the front-most element in the queue.

Q**`.size`**$()$ returns the number of logical elements in the queue.

Figure 3.7: Useful `Queue`-based Methods.

Java PriorityQueue

A *Priority queue* is a rank/score-based data structure wherein the ordering of elements is determined by either their natural ordering or a `Comparator`.

`Queue<`T**`>`** PQ = **`new PriorityQueue<>(`**c**`)`** creates a `PriorityQueue` of type T, named PQ, with a `Comparator` c that is used to compare objects of type T within the priority queue.

PQ**`.add`**(e) inserts e into PQ, whose position in the priority queue depends on the currently-existing elements.

PQ**`.poll`**$()$ returns and removes the element with the highest priority.

PQ**`.peek`**$()$ returns the element with the highest priority.

PQ**`.size`**$()$ returns the number of logical elements in the priority queue.

Figure 3.8: Useful `PriorityQueue`-based Methods.

objects. Therefore when we create a new instance of `Comparator` we must also override its `compare` method. This method's signature varies based on the parameterized type provided to the comparator, but since we want to compare strings, we should declare it as follows:[1]

Listing 3.32

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;

class PriorityQueueByP {

  /**
   * Returns a priority queue that prioritizes strings that start with 'p'.
   * @return priority queue instance.
   */
  static Queue<String> priorityByP() {
    Comparator<String> c = new Comparator<>(
      @Override
      public int compare(String s1, String s2) {
        // TODO.
      }
    );
  }
}
```

We now must specify how to compare `s1` and `s2` to achieve our goal. Strangely enough, if we want to say that `s1` has a higher priority than `s2`, we must return a negative value, similar to the natural ordering of strings (this idea extends to any type we wish to compare, however). So, we perform a case analysis on the strings. If both strings are non-empty, we grab their first character. If both start with 'p', then their ordering depends on a standard lexicographical comparison of the rest of the strings. If the first character of s_1 is 'p', however, we return -1 to designate that s_1 has a higher priority than s_2. Conversely, if s_2 starts with 'p', then we return 1 to designate the opposite. If neither start with p, then again we perform a lexicographical comparison on the entire strings. Algorithm **??** displays the pseudocode for the comparator, but as we will see, we can translate this, verbatim, into Java syntax. The last line of `priorityByP` instantiates a new `PriorityQueue` whose constructor receives the `Comparator` that we just designed.

[1]Again, we are labeling the return type as a *Queue* because priority queues are a type of queue. Upon instantiating a new priority queue, Java will typecheck the return value against the signature of the method to verify that we are returning something that is, in fact, a queue.

Algorithm 1 Pseudocode for Comparing Two Strings For 'p' Priority

```
procedure COMPARE(s1, s2)
    if s1 and s2 are non-empty then
        c1 ← First(s1)
        c2 ← First(s2)
        if c1 is 'p' and c2 is 'p' then
            xs1 ← s1.substring(1)
            xs2 ← s2.substring(1)
            return xs1.compareTo(xs2)
        else if c1 is 'p' then
            return −1
        else if c2 is 'p' then
            return 1
        else
            return s1.compareTo(s2)
        end if
    else
        return s1.compareTo(s2)
    end if
end procedure
```

Listing 3.33

```java
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;

class PriorityQueueByP {

  static Queue<String> priorityByP() {
    Comparator<String> c = new Comparator<>(
      @Override
      public int compare(String s1, String s2) {
        if (!s1.isEmpty() && !s2.isEmpty()) {
          char c1 = s1.charAt(0);
          char c2 = s2.charAt(0);
          if (c1 == 'p' && c2 == 'p') {
            return s1.substring(1).compareTo(s2.substring(1));
          } else if (c1 == 'p') {
            return -1;
          } else if (c2 == 'p') {
            return 1;
          } else {
            return s1.compareTo(s2);
          }
        } else {
          return s1.compareTo(s2);
        }
      }
    );

    return new PriorityQueue<String>(c);
  }
}
```

Let us add a few elements to a priority queue with our custom comparator to exemplify the idea. To add elements, we use `.add`, and to remove the element with the highest priority, we invoke `.poll`.

Listing 3.34

```java
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;

class PriorityQueueByP {

  static Queue<String> priorityByP() {
    // Implementation not shown.
  }

  public static void main(String[] args) {
    Queue<String> pq1 = priorityByP();
    // Add a few values.
    pq1.add("pool");
    pq1.add("peek");
    pq1.add("hello");
    pq1.add("barks");
    pq1.add("park");
    pq1.add("pecking");
    pq1.add("shrub");

    // Poll each from the queue.
    while (!pq1.isEmpty()) {
      System.out.println(pq1.poll());
    }
  }
}
```

The output is as follows:

```
park
pecking
peek
pool
barks
hello
shrub
```

Why is this what the priority queue outputs? If we reason about this with our comparator, it becomes clear. `park` has the highest priority because it starts with 'p' and has a substring that comes before the rest of those strings starting with 'p'. The strings `pecking`, `peek`, and `pool` come next for similar reasons. Finally, none of the strings `barks`, `hello`, and `shrub` start with 'p', so we simply compare based on the strings themselves. The underlying implementation of how the priority queue works is beyond the scope of this textbook. These details are generally reserved for a textbook or course on advanced data structures, which follows the course designed for the audience of this text.

Set-Based Data Structures

Sets are unordered collections of non-duplicate elements. Does this definition sound familiar? It should; it perfectly mirrors the mathematical definition of a set. Java has a few

Java Hash Sets

A *HashSet* is a set of non-duplicate elements with no defined ordering. Elements are inserted into the set by their hashcode.

`Set<`T**`>`** S = **`new HashSet<>()`** creates a `HashSet` of type T, named S.

S**`.contains(`**e**`)`** returns whether or not e is in the set S.

S**`.add(`**e**`)`** adds e to the set S only if it is not present. If e is in S, nothing happens.

S**`.remove(`**e**`)`** removes e from the set S only if it is present. If e is not in S, it returns `false`; otherwise, it returns `true`.

S**`.size()`** returns the number of logical elements in the set.

Figure 3.9: Useful `HashSet`-based Methods.

nuances to its definition of sets that we will now see. We consider these data structures *set-based* since they all rely on the "no-duplicate" philosophy.

`Set` Interface

A *Set* in Java is an interface rather than a class. This is because Java has a hierarchy for differing implementations of sets. We will discuss three: *HashSet*, *TreeSet*, and *LinkedHashSet*. While all three disallow duplicate elements, the latter two impose an ordering on their elements, which goes against the standard mathematical definition, but for practical reasons.

`HashSet` Class

The *HashSet* implementation of sets determines existence of objects in the set by the *hashcode* of the object. Recall that we can think of hashcodes as an association for objects. In fact, in our discussion on strings and arrays, a comparison using == compares the hashcodes of the objects to see if they are identical, rather than containing the same content. Hash sets are convenient and fast to work with due to the underlying implementation, whose details are beyond the scope of this text. Use hash sets when you do not care about element ordering or "position" in the set, but want to ensure no duplicates exist.

`TreeSet` Class

A *TreeSet* is a set with a determined order, either by a natural ordering or that defined by a `Comparator`, like a priority queue. All methods in a `Set` are implemented by a `TreeSet`. Those listed from the `HashSet` figure are also implemented.

`LinkedHashSet` Class

A *LinkedHashSet* is a set with an ordering based on the insertion order of the elements. All methods in a `Set` are implemented by a `LinkedHashSet`. Those listed from the `HashSet` figure are also implemented.

Java Hash Maps

A *HashMap* is a dictionary-based data structure wherein we map *keys* to *values*. The position of these pairs in the map is according to the hashcode of the key.

Map<K, V**>** M **= new HashMap<>()** creates a `HashMap` named M whose keys are of type K and whose values are of type V. Namely, the keys map to the values.

M**.containsKey**(k) returns whether or not k is a key in the map M.

M**.put**(k, v) maps the key k to the value v in M.

M**.get**(k) returns the value associated with k in M, or `null` if k does not have a mapping.

S**.size**() returns the number of logical elements in the set.

Figure 3.10: Useful `HashMap`-based Methods.

Dictionary-Based Data Structures

Dictionaries maps elements from one type K to elements of another type V. These types K and V do not necessarily need to be distinct.

`Map` Interface

Java has an interface called `Map` rather than a class because, like sets, there is a hierarchy for differing implementations of maps. We will discuss three: *HashMap*, *TreeMap*, and *LinkedHashMap*. Maps contain keys and values; the keys are mapped to values in the map. Additionally, maps cannot contain duplicate keys.

`HashMap` Class

HashMaps base existence of keys in the map by their hashcode, identical to a HashSet.

`TreeMap` Class

A *TreeMap* is a map with a determined order, either by a natural ordering of the keys or that defined by a comparator. All method sin a `Map` are implemented by a `TreeMap`.

`LinkedHashMap` Class

A *LinkedHashMap* is a map with an ordering based on the insertion order of the key/value pairs. All methods in a `Map` are implemented by a `LinkedHashMap`. Those listed from the `HashMap` figure are also implemented.

Iterators

We know how to iterate, or traverse, over a simple data structure, e.g., an array. The idea is to use an index and continuously increment the index until we are at the end bounds of the array. Below is a simple example of summing the elements of an array, which we have seen repeatedly by now.

```
static int sum(int[] arr) {
  int sum = 0;
  for (int i = 0; i < arr.size; i++) { sum += arr[i]; }
  return sum;
}
```

The problem is that not all data structures, as we have undoubtedly seen, are sequential; sets and maps are two examples of non-sequential data structures, so how do we traverse over those? Stacks and queues are another example of data structures that are not necessarily sequential. *Iterator* objects are the answer to this problem. Iterators provide a mechanism for traversing over a generalized data structure. Any data structure whose class definition implements `Iterator` must define at least two methods: `boolean hasNext` and `T next`, which determines whether or not we are at the end of the traversal and retrieves the next element respectively. Note that `T`, for the time being, simply means "any type." All of the Java collections implement `Iterator`, we can retrieve the corresponding `Iterator` object via the `.iterator` method.

Upon retrieving an iterator, we can use a `while` loop to continuously traverse over the data structure until no more elements remain to be visited. The elements of the iterator are generated on-the-fly; only upon calling `next` is the value truly read from the data structure itself. Much like the rest of the Collections API, we must pass the parameterized type to the `Iterator` initialization so that it knows what to substitute for `T` in the `next` method.

Example 3.19. Let's use an iterator to traverse over a `LinkedHashSet`, whose elements ordering is determined by their insertion order, meaning the iterator should produce them in the order in which they were inserted.

Listing 3.35—Testing an Iterator over a `LinkedHashSet`

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

import java.util.Set;
import java.util.LinkedHashSet;
import java.util.Iterator;

class IteratorTester {

  @Test
  void testIterator() {
    Set<Integer> lhs = new LinkedHashSet<>();
    lhs.add(8);
    lhs.add(10);
    lhs.add(0);
    lhs.add(90);
    Iterator<Integer> it = lhs.iterator();
    assertAll(
      () -> assertTrue(it.hasNext());
      () -> assertEquals(8, it.next()),
      () -> assertEquals(10, it.next()),
      () -> assertEquals(0, it.next()),
      () -> assertEquals(90, it.next()),
      () -> assertFalse(it.hasNext()));
```

```
  }
}
```

Should we want to traverse over the data again, we need to produce yet another instance of the iterator because there is no way to reset the "position" of an iterator; they are a type of one-time use objects.

It should be stated that the enhanced `for` loop is nearly identical to the job of an `Iterator`, so a programmer may wonder why not use the former over the latter. Inside an enhanced `for` loop, we cannot modify the data structure, meaning that we cannot add, insert, remove, or change elements. On the other hand, iterators allow structural modification. We would not recommend altering the data structure, even if it is permissible by Java, since doing so can result in irksome bugs.

Example 3.20. Let's now iterate over the `LinkedHashSet` using an enhanced `for` loop. We can do so by placing the type on the left-hand side of the element declaration. Our test will simply subtract the elements from left-to-right, since subtraction is not commutative, we can quickly verify the correctness of our result.

Listing 3.36—Using an Enhanced `for` Loop over a Collection

```java
import static Assertions.assertEquals;

import java.util.Set;
import java.util.LinkedHashSet;

class EnhancedForLoopTester {

  @Test
  testEnhancedForLoop() {
    Set<Integer> lhs = new LinkedHashSet<>();
    lhs.add(1);
    lhs.add(2);
    lhs.add(3);
    lhs.add(4);
    int diff = 0;
    for (Integer e : lhs) { diff -= e; }
    assertEquals(-10, diff);
  }
}
```

Streams

Streams are, in effect, a *lazy* collection of "things." By lazy, we mean to say that, if a result is not necessary, or requested, then it is not computed.

Example 3.21. Consider a situation in which we invoke a method called `omega()`, which is defined as an infinite loop, as the argument to the `foo` method, giving us `foo(omega())`. In Java, all arguments are evaluated *eagerly*, which means that, in effect, takes an eternity to terminate. Unfortunately for the caller of `foo`, we do not even use the value of x, meaning we computed `omega()` for absolutely no reason whatsoever. This means that, because `omega` never terminates, `foo` similarly never returns a result.

```java
static int omega() {
  while (true) {}
  return 10;
```

```
}
static int foo(int x) {
  return 5;
}
```

If Java supported *lazy evaluation* for method calls, we would not be in this predicament. Our discussion is not entirely driven by a desire for lazy evaluation, but rather the desire for easily-composable operations; lazy evaluation is a perk in that it allows us to design infinite data structures! An "infinite" data structure raises some important questions about how to store "infinite" data. Imagine that we want to compute a list that contains every positive even integer. We can represent this as the following inductive set:

$$0 \in S$$
$$\text{If } x \in S, \text{ then } x + 2 \in S$$

Therefore S is a set containing countably-infinite values. Implementing S in Java, as an `ArrayList`, might contain a `for` loop with a condition that we do not know how to solve! In this case, since we do not know how many values to add, we might design an infinite loop via `while (true)`, but then the loop never ends. Eventually the program runs out of memory due to adding values to the never-ending list. The solution, as we have suggested, is to use streams.[1]

To create a stream of infinite data is to recreate our inductive set definition inside a `IntStream` instance and the `iterate` static method.

```
IntStream is = IntStream.iterate(0, x -> x + 2);
```

Let us explain this method, but to do so we must introduce *lambda expressions.* A lambda expression is an anonymous function, i.e., a function definition without a name. In the above code snippet, we define a function that receives a value x and returns x plus two. It would be identical to defining a `private static` method to add two to some integer, but we like lambda expressions due to their locality; it might come across as superfluous to design a method that is used in only one context. Should we want to pass a method reference instead of a lambda expression, this is easily attainable.

Listing 3.37

```java
import java.util.IntStream;

class PositiveEvens {

  private static int addTwo(int x) { return x + 2; }

  public static void main(String[] args) {
    IntStream is = IntStream.iterate(0, PositiveEvens::addTwo);
  }
}
```

The `IntStream` instance declares a stream that, when requested/prompted, invokes and populates the stream. Because it is impossible to represent an infinite data structure in Java with modern computers, we should limit how many values we want from this stream. Indeed, the `.limit` method computes exactly n elements from the stream. So, to compute the first ten elements of our *is* `IntStream`, we invoke `.limit(10)` on our *is* stream.

[1]For those coming from another language such as Python, a stream is equivalent to a *generator*.

Listing 3.38

```
import java.util.IntStream;

class PositiveEvens {

  public static void main(String[] args) {
    IntStream is = IntStream.iterate(0, x -> x + 2).limit(10);
  }
}
```

Now, suppose we want to view these ten elements. Right now they are consolidated into an `IntStream`, but we need to convert them to a list of sorts. The solution is to convert the values into a `Stream<Integer>` via `.boxed()`, and then to a list using the convenient `.toList()` method.

Listing 3.39

```
import java.util.IntStream;
import java.util.List;

class PositiveEvens {

  public static void main(String[] args) {
    IntStream is = IntStream.iterate(0, x -> x + 2).limit(10);
    List<Integer> ls = is.boxed().toList();
    System.out.println(ls); // [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
  }
}
```

Example 3.22. Suppose we want a stream of infinitely repeating `"a"` strings. We can easily do this via the `generate` method, which acts as the stream constructor, receiving a lambda expression to continuously generate new elements.

Listing 3.40

```
import java.util.Stream;

class AGenerator {

  public static void main(String[] args) {
    Stream<String> as = Stream.generate(() -> "a");
    // ["a", "a", "a", "a", "a", "a", "a", "a", "a", "a"]
    System.out.println(as.limit(10).boxed().toList());
  }
}
```

Again, it is important to understand what is going on under the hood of a stream. Elements thereof only generate when we request them through some accessory means, e.g., `limit`. As we previously suggested, attempting to access an infinite stream without a limit causes the program to hang and eventually crash with an `OutOfMemoryError` exception.

Example 3.23. Imagine we want to create a stream of all of the Fibonacci numbers. The thing is, eventually we will reach the 32-bit limit for the `int` datatype, so we should take advantage of the `BigInteger` class, which allows us to represent arbitrarily-large integers.[1] Also, this time we will write a method that returns the stream instance rather than creating it in the `main` method.

Here's what we need to do: we will use `iterate` to generate new values in the sequence. There is a slight problem in that the Fibonacci sequence has two starting (accumulator) values: 0 and 1. The issue is that `iterate` receives only one "initializer" value. To circumvent this predicament we can simply pass an array that contains the current and "next" Fibonacci values. Inside the lambda expression we of course receive an array of values, from which we can compute the next Fibonacci number. This time, however, instead of using `IntStream`, we will generalize to the `Stream` class since our initial value(s) is not an integer.

Listing 3.41

```java
import static Assertions.assertEquals;

import java.util.Stream;
import java.util.List;
import java.util.ArrayList;
import java.util.BigInteger;

class BigIntFibStreamTester {

  @Test
  void bigIntFibStreamTest() {
    // Get the stream, test ten values, make sure the lists are the same
    // length, then test each subarray.
    Stream<BigInteger[]> s = StreamExample.fibonacciStream();
    List<BigInteger[]> actualLs = s.limit(10).toList();
    List<BigInteger[]> expectedLs
      = new ArrayList<>(
        List.of(new BigInteger[]{new BigInteger("0"), new BigInteger("1")},
                new BigInteger[]{new BigInteger("1"), new BigInteger("1")},
                ...));

    // Check each array of BigIntegers of the expected and actual.
    assertTrue(expectedLs.size() == actualLs.size());
    for (int i = 0; i < expectedLs.size(); i++) {
      assertArrayEquals(expectedLs.get(i), actualLs.get(i));
    }
  }
}
```

Listing 3.42

```java
import java.util.Stream;
import java.util.BigInteger;

class BigIntFibStream {

  /**
   * Computes a stream of BigInteger values computing the nth Fibonacci value.
   * @return stream containing arrays of the next sequential BigIntegers.
   */
  static Stream<BigInteger[]> fibonacciStream() {
    BigInteger[] vals = new BigInteger[]{new BigInteger("0"),
```

[1] Worrying about *how* the `BigInteger` class works for now is unnecessary as our current plan is to demonstrate stream properties.

Java Streams

A *stream* is a lazy collection of elements that are computed only when requested.

`int` S**`.count()`** returns the number of elements in the stream.

`Stream<`T**`>`** S**`.map(`**f**`)`** returns a new stream whose elements are the result of applying f to each element of S.

`Stream<`T**`>`** S**`.filter(`**p**`)`** returns a new stream of values in S that satisfy the predicate p.

`Stream<`T**`>`** S**`.reduce(`**a**`,`** f**`)`** returns the result of applying the binary function f to each element of S, starting from a, which serves as the accumulator's initial value.

`Stream<`T**`>`** S**`.limit(`**n**`)`** returns a new stream containing the first n elements of S.

`Stream<`T**`>`** S**`.skip(`**n**`)`** returns a new stream containing the elements of S after the first n.

`Optional<`T**`>`** S**`.min/max(`**c**`)`** returns the minimum/maximum element of S according to the comparator c. If S is empty, returns `Optional.empty()`.

Figure 3.11: Useful `Stream`-based Methods.

```
                                        new BigInteger("1")};
    return Stream.iterate(vals, v -> new BigInteger[]{v[1], v[0].add(v[1])});
  }
}
```

So our code now produces a list of `BigInteger` arrays containing the current Fibonacci value and its successor. Though, is this really what we want? A better solution would be to simply return the first element of the tuple/two-element array. We can achieve this via the `map` function. `map` receives a lambda expression as an argument and applies it to every element of the acting stream. Let's modify the code a bit to see an improved output. Excellently, this change means we do not need to loop over our expected/actual lists in the unit testing method, as `assertEquals` works as intended over `List` objects.

Listing 3.43

```
import static Assertions.assertEquals;

import java.util.Stream;
import java.util.List;
import java.util.ArrayList;
import java.util.BigInteger;

class BigIntFibStreamTester {

  @Test
  void testBigIntFibStream() {
    Stream<BigInteger> s = StreamExample.fibonacciStream();
    List<BigInteger> actualLs = s.limit(10).toList();
    List<BigInteger[]> expectedLs
      = new ArrayList<>(
        List.of(new BigInteger[]{new BigInteger("0"), new BigInteger("1")},
                new BigInteger[]{new BigInteger("1"), new BigInteger("1")},
```

Java Stream–Searching Methods

We can search for the existence of types of elements in a stream.

boolean S**.anyMatch(**p**)** returns `true` if **at least one** element of S satisfies the predicate p; otherwise, returns `false`.

boolean S**.allMatch(**p**)** returns `true` if **all** elements of S satisfy the predicate p; otherwise, returns `false`.

boolean S**.noneMatch(**p**)** returns `true` if **no** elements of S satisfy the predicate p; otherwise, returns `false`.

Figure 3.12: Useful `Stream`-Searching Methods.

```
            ...));
    assertEquals(expectedLs, actualLs);
  }
}
```

Listing 3.44

```
import java.util.Stream;
import java.util.BigInteger;

class BigIntFibStream {

  /**
   * Computes a stream of BigInteger values computing the nth Fibonacci value.
   * @return stream containing arrays of the next sequential Fibonacci BigIntegers
   *   .
   */
  static Stream<BigInteger> fibonacciStream() {
    BigInteger[] vals = new BigInteger[]{new BigInteger("0"),
                                         new BigInteger("1")};
    return Stream.iterate(vals, v -> new BigInteger[]{v[1], v[0].add(v[1])})
                 .map(v -> v[0]);
  }
}
```

We will now take a bit more of a look at `map`, as well as other useful *higher-order functions* such as `filter` and `reduce`.

A *higher-order function* is a function that takes functions as parameters. We saw that `map` receives a lambda expression and applies it to every element of a stream.

Example 3.24. Let's write the `sqList` method that receives a `List<Integer>` and squares each element using the Stream API. The method should return a new list. A motif presented throughout stream methods is that they do not modify the original data. We should use `map` to apply a lambda expression that receives an integer and returns its square. Fortunately for us, we can convert any collection into a stream using the `.stream()` method. From there, we use a simple `map` invocation to arrive at our desired outcome.

Listing 3.45

```java
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static SqList.sqList;

import java.util.List;
import java.util.ArrayList;

class SqListTester {

  @Test
  void sqListTest() {
    List<Integer> ls1 = new ArrayList<>(List.of(1, 4, 9, 16, 25));
    List<Integer> ls2 = new ArrayList<>(List.of(0, 100, 81, 81));
    List<Integer> ls3 = new ArrayList<>();
    assertAll(
      () -> assertEquals(ls1, sqList(new ArrayList<>(List.of(1, 2, 3, 4, 5)))),
      () -> assertEquals(ls2, sqList(new ArrayList<>(List.of(0, 10, 9, 9)))),
      () -> assertEquals(ls3, sqList(new ArrayList<>())));
  }
}
```

Listing 3.46

```java
import java.util.List;

class SqList {

  /**
   * Returns a list of squared integers from a list of integers.
   * @param list of integers.
   * @return list of squared integers.
   */
  static List<Integer> sqList(List<Integer> ls) {
    return ls.stream()
             .map(x -> x * x)
             .toList();
  }
}
```

Example 3.25. Now, let's write the `removeVowels` method, which receives a string and removes all vowels, returning a new string in the process. This requires a few techniques that we have learned, but also means we need to use `filter` and `reduce`. Here's the idea:

1. Convert the given `String` into a stream of integers representing the ASCII values of characters.
2. Convert each integer to a "one-string," i.e., a `String` of one character. The reasoning behind this decision will become clear later.
3. Filter out vowels from the stream.
4. Accumulate the characters in a new string.

Of course, we begin by writing a few tests. Fortunately, writing tests is simple for this method, whereas the implementation is the most complex seen thus far.

Listing 3.47—Tests for `removeVowels` Method

```
import static Assertions.assertAll;
import static RemoveVowelsStream.removeVowels;

class RemoveVowelsStreamTester {

  @Test
  void testRemoveVowels() {
    assertAll(
      () -> assertEquals("hll", StreamTests.removeVowels("hello")),
      () -> assertEquals("hw r y?", StreamTests.removeVowels("how are you?")),
      () -> assertEquals("", StreamTests.removeVowels("aaaaaaaaaaaa")),
      () -> assertEquals("bbbbbbbbbb", StreamTests.removeVowels("bbbbbbbbbb")),
      () -> assertEquals("bbbbb", StreamTests.removeVowels("ababababa")),
      () -> assertEquals("hll", StreamTests.removeVowels("aeiouAEIOU")));
  }
}
```

Onto the definition; we start by writing the method signature and purpose. Then, we need to complete step one: convert the given string into a stream of ASCII integers, which is achievable via the `.chars` method. It returns an `IntStream` of integer ASCII values. Step two involves us converting each integer into a "one-string," which we can do via the constructor for a `String` object. Step three requires the use of `filter`, which is another higher-order function. It receives a lambda expression and returns those objects from the stream that satisfy the filter. Since we want to filter *out* the vowels, we should write a method that determines if a character is vowel, then negate the expression as part of the lambda definition. Lastly we arrive at accumulating the characters into a new string, requiring us to use `reduce`: yet another higher-order function. The reduce function receives an initial value, i.e., an accumulator a and a binary function f. It then applies the binary function to each value in the stream and the running accumulator. If this reminds you of tail recursion, then indeed, that is exactly how reduce works; It folds over the list/stream of values, building the result in the accumulator variable.[1] Due to the simplicity of the `isVowel` predicate and its implementation in Chapter 2, we omit its definition.

Listing 3.48—Definition of `removeVowels` Using Streams

```
class RemoveVowelsStream {

  /**
   * Removes all vowels from a given string using streams.
   * @param s - string to remove vowels from.
   * @return new string.
   */
  static String removeVowels(String s) {
    return s.chars()
            .mapToObj(c -> String.valueOf(c))
            .filter(c -> !isVowel(c))
            .reduce("", (acc, c) -> acc + c);
  }
}
```

[1] In other functional programming languages, `reduce` is commonly called `foldr`.

Optional Type

The primary benefit of streams is their compositionality. We can chain together multiple operations, sequentially, to compute a result. Though, there are instances in which a value may not exist, and the stream has to account for these somehow.

Example 3.26. Consider a series of stream operations to find the maximum integer of a list. For the general case, this is straightforward, but what about when the list is empty? It does not make sense to return zero, since the maximum integer in a list may very well be zero, which leads to a false conclusion. The solution, in this case, is the `Optional` class. An `Optional` is a container that may or may not contain a value. If so, we may access the value directly via `.get`. If we do not know whether or not it contains a value, we may use `.orElse(`t`)`, which returns the encapsulated value if it exists, or t otherwise. We can also check, prior to a retrieval, if the `Optional` contains a value via the `.isPresent` method. `Optional` is generic and works over any class type, like almost all other classes from the collections API. To test the return value of a stream operation that returns an `Optional`, e.g., `max`, we instantiate an `Optional` that wraps the resulting value if it exists, or `Optional.empty()` otherwise.

Listing 3.49—Using the Optional Wrapper Class

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

import java.util.Optional;
import java.util.List;

class OptionalTester {

  private static final List<Integer> LS1 = List.of(10, 20, 42, 12, 5);
  private static final List<Integer> LS2 = List.of();

  @Test
  void testMaxValue() {
    Optional<Integer> op1 = Optional.of(42);
    Optional<Integer> op2 = Optional.empty();
    assertAll(
      () -> assertEquals(op1, LS1.stream().max((a, b) -> a - b)),
      () -> assertEquals(op2, LS2.stream().max((a, b) -> a - b)),
      () -> assertEquals(42, LS1.stream().max((a, b) -> a - b).orElse(null));
      () -> assertNull(LS2.stream().max((a, b) -> a - b).orElse(null)));
  }
}
```

Optional values, like we stated, work wonders with the compositionality of streams; if a value does not exist, the stream API will propagate an `empty` instance of `Optional` up the chain rather than displaying an error or crashing the program. As part of the design philosophy of the class, those decisions, i.e., whether to crash the program or not, remain a choice left for the implementing programmer.

3.3 Type Parameters

Generics as a concept go far back in programming history, generally reknown as type parameterization, which we briefly touched on during our discussion of how to instantiate instances of `ArrayList` from the Collections API. Imagine, for a moment, if the Java programmers had to write a differing implementation of `ArrayList` for `Integer`, `String`, `Double`, and so on ad infinitum. Not only is this impossible, it would also be extremely cumbersome and redundant, since the only altering parameter is the underlying element type in the data structure. Before Java 5, we could only use "generics" via collections of type `Object`, since it is the root class object type, meaning any element could be stored in any type of collection.

Listing 3.50—Pre Java-5 "Generics" Example

```java
import java.util.ArrayList;

class WeakGenerics {

  public static void main(String[] args) {
    ArrayList al1 = new ArrayList();
    al1.add(new Integer(42));
    al1.add(new Integer(43));
    Integer x = (Integer) al1.get(0);
  }
}
```

Performing casts like this is prone to errors, not to mention the possibility of adding disjoint types into a collection. For example, there is nothing preventing us from adding objects of type `String` or `Integer` into an `ArrayList` at this time. Generics were introduced to convert the problem from one encountered at runtime to one encountered moreso at compile-time.

Since we have yet to discuss objects in detail, we will hold off on a significant discussion of generic class implementations. To keep it to the point, we can write any class to be generic and store objects of an arbitrary type. Fortunately we can also do the same with static methods.

To declare a static method as generic, we must specify the type variable(s) necessary to use the method. These come after the `static` keyword but before the return type. For instance, if we want to say that an object of type `T` is used in the static method `foo`, we declare it as `static <T> void foo(...){...}`. Then, if we want to say that the method returns or receives an object of type `T`, we substitute the return/parameter type with `T`, e.g., `static <T> T foo(T arg){...}`. At compile-time, Java will look for method invocations of `foo` and substitute the `T` for whatever type `foo` is invoked.

Example 3.27. Let's design the `int search(List<T> t, T k)` method, which receives a list t and an object k, where the elements of t are of type `T` and k is also of type `T`. Its purpose is to return the index of the first occurrence of k in t, and -1 if it does not exist. Because all objects have `.equals`, we can take advantage of this fact when comparing objects in the list against the search parameter. When testing, we will instantiate the type parameter to several different types to demonstrate.

Listing 3.51—Testing the Generic Search Method

```java
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static GenericSearch.genericSearch;

import java.util.List;
```

```java
import java.util.ArrayList;

class GenericSearchTester {

  @Test
  void genericSearchTest() {
    List<Integer> l1 = new ArrayList<>(List.of(1, 2, 3, 4, 5));
    List<String> l2 = new ArrayList<>(List.of("a", "b", "c", "d", "e"));
    List<Double> l3 = new ArrayList<>(List.of(1.0, 2.0, 3.0, 4.0, 5.0));
    List<Character> l4 = new ArrayList<>(List.of('a', 'b', 'c', 'd', 'e'));
    List<List<Integer>> l5 = new ArrayList<>();

    assertAll (
      () -> assertEquals(1, genericSearch(l1, 2)),
      () -> assertEquals(-1, genericSearch(l1, 6)),
      () -> assertEquals(1, genericSearch(l2, "b")),
      () -> assertEquals(-1, genericSearch(l2, "f")),
      () -> assertEquals(1, genericSearch(l3, 2.0)),
      () -> assertEquals(-1, genericSearch(l3, 6.0)),
      () -> assertEquals(1, genericSearch(l4, 'b')),
      () -> assertEquals(-1, genericSearch(l4, 'f')),
      () -> assertEquals(-1, genericSearch(l5, List.of(1, 2, 3))));
  }
}
```

Listing 3.52—Generic Search Method

```java
import java.util.List;

class GenericSearch {

  /**
   * Returns the index of the first occurrence of k in t,
   * or -1 if it does not exist.
   * @param t the list of type T.
   * @param k the object of type T to search for.
   * @return the index of k or -1.
   */
  static <T> int genericSearch(List<T> t, T k) {
    for (int i = 0; i < t.size(); i++) {
      if (t.get(i).equals(k)) { return i; }
    }
    return -1;
  }
}
```

Bounded Type Parameters

To restrict the type parameter to a certain subset of types, we can use *bounded type parameters*. As an example, we might wish to restrict a type parameter for a method to only types that implement the `Comparable` interface. Doing so means that the type parameter has access to any methods defined by the interface, in this case, `compareTo` being the only available method. To specify a bounded type parameter, we use the `extends` keyword, e.g., `<T extends Comparable>`. We denote this as an upper-bound on the type parameter, since we are restricting the type parameter to a subset of types that are "above" the specified type. We might also wish to use a lower-bound, which restricts the type parameter to a subset of types that are "below" the specified type. For example, if we want to restrict the type parameter to only types that are superclasses of `Integer`, we can do so by specifying `<T super Integer>`.

Example 3.28. Let's design the `static <T extends Comparable<T>> T max(List<T> t)` method, which receives a list *t* of type `T` and returns the maximum element in the list. Because determining the max element of a list involves comparison-based checking, we must restrict the type parameter to only types that implement an interface for comparing objects, e.g., `Comparable`. In the previous section we discussed that `Optional` is a container class that can either hold a value or be empty. An exercise at the end of this section will ask you to use `Optional` in designing a similar method, rather than returning `null` as we will show here.

Listing 3.53—Testing the Generic Max Method

```java
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static GenericMax.genericMax;

import java.util.List;
import java.util.ArrayList;

class GenericMaxTester {

  @Test
  void genericMaxTest() {
    List<Integer> l1 = new ArrayList<>(List.of(5, 10, 20, 7, 2));
    List<String> l2 = new ArrayList<>(List.of("A", "e", "x", "Z", "3", "N"));
    List<Double> l3 = new ArrayList<>(List.of(500.0, 400.0, 3.0, Math.PI, 200.0));
    List<Character> l4 = new ArrayList<>(List.of('?','@','A','a','Z'));
    List<List<Integer>> l5 = new ArrayList<>();

    assertAll (
      () -> assertEquals(20, genericMax(l1)),
      () -> assertEquals("x", genericMax(l2)),
      () -> assertEquals(500.0, genericMax(l3)),
      () -> assertEquals('a', genericMax(l4)),
      () -> assertEquals(null, genericMax(l5)));
  }
}
```

Listing 3.54—Generic Max Method

```java
import java.util.Comparable;
import java.util.List;

class GenericMax {

  /**
   * Returns the maximum element in the list according to
   * the compareTo implementation of the type parameter.
   * @param t the list of type T, where T is a type that implements Comparable.
   * @return the maximum element in the list.
   */
  static <T extends Comparable<T>> T genericMax(List<T> t) {
    if (t.isEmpty()) { return null; }
    else {
      T max = t.get(0);
      for (int i = 1; i < t.size(); i++) {
```

```
        if (t.get(i).compareTo(max) > 0) {
          max = t.get(i);
        }
      }
      return max;
    }
  }
}
```

Wildcards and Unspecific Bounds

Sometimes we want to specify that a collection contains different, but related, types. If we declare a `List<Integer>`, then Java throws a compile-time error if we attempt to pass, say, an unboxed `double`, since a `Double` is not of type `Integer`. As a solution, Java allows the programmer to enforce *wildcard* bounds on the type of a generic.

Example 3.29. Reusing the example that we just talked about, if we want to store both `Integer` and `Double` objects in the same collection, we need to look to see how they are related. Both of these classes extend the `Number` superclass, so we can place an upper-bound on the type parameter to say that anything that extends `Number`, whatever it may be, can be stored in the collection. Wildcards are denoted via the question mark: '?', to represent that it can be substituted with any other type that meets the bound criteria. Our example method, which computes the sum of a list of numbers, requires the substitutable type to be a subclass of `Number`.

Listing 3.55

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

import java.util.List;

class NumberListTester {

  private static final double DELTA = 0.0000001;

  @Test
  void testSum() {
    assertAll(
      () -> assertEquals(0, NumberList.sum(List.of())),
      () -> assertEquals(42,
                         NumberList.sum(
                           List.of(-0.2f, -1, (short) 43, 2.6d,
                                   (byte) 100, -100L, -2.4f)),
                         DELTA));
  }
}
```

Listing 3.56

```
import java.util.List;

class NumberList {

  /**
   * Computes the sum of a list of Number subclasses.
   * @param ls list of Number instances.
   * @return sum as a double.
```

```
   */
  static double sum(List<? extends Number> ls) {
    return ls.stream()
             .mapToDouble(Number::doubleValue)
             .sum();
  }
}
```

Example 3.30. Lower-bounded type parameters are the dual to upper-bounded type parameters. If we want to restrict our possible types in a generic implementation to be only superclasses of a type, we can easily do so. For instance, the following code specifies that the input list can only contain objects that are superclasses of `Integer`, or are `Integer` itself. Unfortunately, the only classes that are ancestors/superclasses of `Integer` are `Number` and `Object`, which severely limits what we can do with our list. For the sake of an example, we might return a string containing the "stringified" elements, separated by commas, enclosed by brackets.

Listing 3.57

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

import java.util.List;

class NumberListTester {

  @Test
  void testStringify() {
    assertAll(
      () -> assertEquals("[]", NumberList.stringify(List.of())),
      () -> assertEquals("[42, 32, Hi]",
                        NumberList.stringify(List.of(42, 32, (Object) "Hi"))));
  }
}
```

Listing 3.58

```
import java.util.List;
import java.util.stream.Collectors;

class NumberList {

  /**
   * Receives a list of objects that are superclasses of Integer
   * and converts them to their string counterparts and puts them
   * in a list representation.
   * @param ls list of instances that are superclasses of Integer.
   * @return stringified list.
   */
  static String stringify(List<? super Integer> ls) {
    return ls.stream()
             .map(Object::toString)
             .collect(Collectors.joining(", ", "[", "]"));
  }
}
```

Without our own classes to work with, the potential benefits for wildcards and type parameter bounds are not easy to spot. We present them in this chapter, though, to show that they at least exist in Java.

Chapter Exercises

Exercise 3.1. (⋆)
Design the `int[] operate(int[] A)` method that, when given an array of integers, returns a new array where the elements are the result of applying the following operation to each element: if the i^{th} element is odd, multiply it by its index i plus one. Otherwise, divide it by its index i plus one.

Exercise 3.2. (⋆)
Design the `boolean containsOnlyPrime(int[] arr)` method that returns whether or not a given array of integers contains only prime integers. Hint: use the method you wrote in Chapter 2.

Exercise 3.3. (⋆)
This question has two parts.

(a) Design the recursive `linearSearch` method that, when given a `String[]` S and a `String` to search for k, returns the index of k in S, and `-1` if k is not in the array. This method is definitionally tail recursive, you should write a `private` helper method that actually performs the recursion.[1]

(b) Design the `linearSearchLoop` method that solves the problem using a loop.

Exercise 3.4. (⋆⋆)
Design the `int[] accSum(int[] A)` method, which receives an array of integers and returns a new array of integers that corresponds to the accumulated sum between each integer. We present some examples below.

```
accSum({1, 7, 2, 9})                => {1, 8, 10, 19}
accSum({1, 3, 3, 4, 5, 5, 6, 6, 2}) => {1, 4, 7, 11, 16, 21, 27, 33, 35}
accSum({5, 5, 5, 5, 5, 5, 5, 1, 5}) => {5, 10, 15, 20, 25, 30, 35, 36, 41}
```

Exercise 3.5. (⋆)
Design the `String[] fizzBuzz(int min, int max)` method that iterates over the interval [*min*, *max*] (you may assume *max* $\geq$ *min*) and returns an array containing strings that meet the following criteria:

- If i is divisible by 3, insert `"Fizz"`.
- If i is divisible by 5, insert `"Buzz"`.
- If i is divisible by both 3 and 5, insert `"FizzBuzz"`.
- Otherwise, insert `"i"`, where `i` is the current number.

```
fizzBuzz(1, 12)  => {"1", "2", "Fizz", "4", "Buzz",
                     "Fizz", "7", "8", "Fizz", "Buzz",
                     "11", "Fizz"}
fizzBuzz(15, 18) => {"FizzBuzz", "16", "17", "Fizz"}
```

Exercise 3.6. (⋆⋆)
Using only arrays, design the `int[] findIntersection(int[] A, int[] B)` method that returns an array containing the *intersection* of two arrays. The intersection of two arrays is the set of elements that are common to both arrays. For example, the intersection

[1] By "definitionally tail recursive," we mean that, even though a standard recursive variant exists, it is strongly recommended to only use a tail recursive algorithm, given the recursive definition requirement.

of $\{7, 4, 8, 0, 2, 1\}$ and $\{8, 6, 4, 9, 26, 4, 0\}$ is $\{7, 8, 0\}$. Do not assume that the arrays are sorted, and you cannot sort them yourself.

Exercise 3.7. (⋆⋆)
Desing the `int median(int[] A, int[] B)` that, when given two sorted (in increasing order) arrays of integers A and B, returns the median value of those two lists. You can use auxiliary data structures to help in solving the problem, but they are not necessary.

Exercise 3.8. (⋆)
Design the `double sumNasty(ArrayList<Integer> vals)` method that returns the average of the numbers in the list with the following caveat: The number 9 should not be counted towards the average, nor should the following number, should one exist.

```
sumNasty({8, 7, 11, 9, 12, 10}) => 9.0
sumNasty({120, 99, 9})          => 109.5
sumNasty({9})                   => 0.0
sumNasty({})                    => 0.0
```

Exercise 3.9. (⋆)
Design the `int[] countEvenOdds(int[] vals)` method that returns a tuple (an array of two values) where index zero stores the amount of even values and index one stores the amount of odd values.

```
countEvenOdds({11, 9, 2, 3, 7, 10, 12, 114}) => {4, 4}
countEvenOdds({11, 13, 15, 17})              => {0, 4}
```

Exercise 3.10. (⋆⋆)
This question has two parts.

(a) Design the `isAlmostStrictlyIncreasing` tail recursive method that, when given an array of integers, determines if it is strictly increasing. There is a catch to this: we also return true if the array can be made strictly increasing by removing exactly one element from the array. For instance, `isAlmostStrictlyIncreasing`($\{1, 3, 2, 4\}$) returns true because, by removing 3, we get a list that is strictly increasing. Compare this with `2, 3, 2, 4`, which cannot be made strictly increasing.

(b) Design the `isAlmostStrictlyIncreasingLoop` method that solves the problem using a loop.

If you write tests for one of these methods, you should be able to propagate it through the rest, so write plenty!

Exercise 3.11. (⋆⋆)
Design the `boolean isSubArray(int[] A, int[] B)` method, which receives two arrays of integers A and B, and determines if B is a "sub-array" of A. This means that all elements of B are elements of A.

Exercise 3.12. (⋆⋆)
Design the `int[] twoDimToOneDim` (int[][] A) method that flattens a given two-dimensional array of integers into a one-dimensional array. Hint: figure out how many elements the resulting one-dimensional array should have, and only then figure out how to position elements.

Exercise 3.13. (⋆⋆⋆)
Design the `boolean canBalanceArray(int[] A)` method, which determines whether or not an array of integers A can be split into a partition that balances each side. Use the following examples as motivation.

```
canBalanceArray(new int[]{1, 1, 1, 1, 5})    => false
canBalanceArray(new int[]{2, 3, 5})          => true
canBalanceArray(new int[]{-10, 2, -58, 50})  => true
canBalanceArray(new int[]{3, -1, -1, -1, 0}) => true
canBalanceArray(new int[]{3, 2, 1, 0})       => true
canBalanceArray(new int[]{10})               => false
```

Exercise 3.14. (⋆⋆⋆)
A *span* is the distance between a value and another, distinct occurrence of the same value. Design the `int largestSpan(int[] A)` method that, when given a non-empty array of integers A, returns the largest span. It may be beneficial to write the `firstIndexOf` and `lastIndexOf` methods to help in your design process. Use the following examples as motivation.

```
largestSpan(new int[]{4, 2, 3, 2, 5})             => 3
largestSpan(new int[]{1, 2, 3, 4, 5})             => 1
largestSpan(new int[]{5, 4, 4, 4, 3, 4, 1, 4, 1}) => 7
```

Exercise 3.15. (⋆⋆)
We can represent matrices as two-dimensional arrays. For example, the matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

can be represented as the two-dimensional array

```
int[][] matrix = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

Design the `int[][] transpose(int[][] matrix)` method that returns the transpose of a given matrix. The transpose of a matrix is the matrix where the rows and columns are swapped. For example, the transpose of the above matrix is

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Exercise 3.16. (⋆⋆)
Design the `int[][] rotate(int[][] matrix)` method that returns the matrix rotated 90 degrees clockwise. For example, the matrix

$$\begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

is rotated to

$$\begin{bmatrix} 3 & 2 & 1 \\ 6 & 5 & 4 \\ 9 & 8 & 7 \end{bmatrix}$$

To rotate a matrix, take its transposition, then reverse each row. You may assume that the input matrix is $N \times N$, i.e., a square matrix.

Exercise 3.17. (⋆⋆)
Design the `int[][] multiply(int[][] A, int[][] B)` method that returns the product of two matrices, where A is $M \times N$ and B is $P \times Q$. Not all matrices can be multiplied, so you should return `null` if the matrices cannot be multiplied. Two matrices can be multiplied if and only if $N = P$. The product of two matrices A and B is the matrix C where $C_{i,j} = \sum_{k=1}^{N} A_{i,k} \cdot B_{k,j}$ for the indices i, k, and j.

Exercise 3.18. (⋆⋆)
Design the `boolean canSum(int[] A, int t)` method that, when given an array of integers A and a target t, determines whether or not there exists a gruop of numbers in A that sum to t. For example, if $A = \{2, 4, 10, 8\}$ and $t = 9$, then `canSum` returns false because there is no possible selection of integers from A that sum to 9. On the other hand, if $A = \{3, 7, 4, 5, 9\}$ and $t = 8$, then we return true because $3 + 5 = 8$. If $A = \{2, 4, 2, 11, 5, 4\}$ and $t = 9$, then we return true because $1 + 4 + 4 = 9$, but also $4 + 5 = 9$ and $5 + 4 = 9$.

There is a simple recursive algorithm to solve this problem: if you have searched through the entire array, return whether or not the target is zero. Otherwise, make two recursive calls: one for where you choose the current number and a second for where you do not. By "choose," we mean to say that the current number is subtracted from the target value. By "current number," we mean to suggest a design that resemble the tail recursive linear search algorithm. You'll need to design a helper method to solve this problem using this approach.

Exercise 3.19. (⋆⋆)
Design the `List<Integer> sumEvenMultOdd(int[] A)` method that, when given an array of integers A, returns a `List<Integer>` whose first element is the sum of all elements at even indices of A, and whose second element is the product of all elements at odd indices of A. If you encounter a zero at an odd index, do not continue to multiply values (i.e., don't keep multiplying subsequent values and return the multiplied value before you encountered the zero, since the product will forever be zero from that point onward).

Exercise 3.20. (⋆⋆)
Design the `Set<List<Integer>> twoSum(int[] A, int t)` method that, when given an array of integers A and a target t, returns all possible pairs of numbers in A that sum to t. For example, if $A = \{2, 2, 4, 10, 6, -2\}$ and $t = 4$, we return a set containing two two-element lists: $\{2, 2\}$ and $\{6, -2\}$. DO not add a pair that already exists in the set or a pair that, by reversing the pair, we get a pair in the existing set. E.g., $\{-2, 6\}$ should not be added to the set.

There is a simple brute-force algorithm to solve this problem via two loops, but by incorporating a second set for lookups, we can do much better: for every number n in A, add n to a set S, and if $t - n = m$ for some $m \in S$, then we know that $m + n$ must equal t, therefore we add $\{n, m\}$ to the resulting set of integer arrays. Walking through this with the example from before, we get the following sequence of actions:

- Initialize $S = \{\}$ and $L = \{\}$. We know that $t = 4$.
- We add 2 to S. $S = \{2\}$.
- Because $4 - 2 \in S$, the two-element array $\{2, 2\}$ is added to L. 2 is not re-added to S.
- Because $4 - 4 \notin S$, we only add 4 to S. $S = \{2, 4\}$.

- Because $4 - 10 \notin S$, we only add 10 to S. $S = \{2, 4, 10\}$.
- Because $4 - 6 \notin S$, we only add 6 to S. $S = \{2, 4, 10, 6\}$.
- Because $4 - (-2) \in S$, the two-element array $\{6, -2\}$ is added to L. We add -2 to S. $S = \{2, 4, 10, 6, -2\}$.

Exercise 3.21. ($\star\star$)
Design the `ArrayList<String> shift(ArrayList<String> ls, int i)` method that, when given a list of strings s and an index i, returns a new list where each element is shifted by i spots. Negative values correspond to left shifts, and positive values correspond to right shifts. If a shift is nonsensical, do not shift at all. A nonsensical shift is one of the following:

- If there are no elements in the list.
- If there is only one element in the list.
- If there are only two elements in the list, you only need to shift once. Do the math!

This method is harder than it may appear at first glance, so write plenty of tests!

```
shift({11, 12, 13, 14}, -1)          => {12, 13, 14, 11}
shift({120, 120, 140, 140}, 2)       => {140, 140, 120, 120}
shift({99999999}, 1000)              => {99999999}
shift({}, -9999999)                  => {}
shift({120, 80, 70, 50, 40, 20}, -3) => {50, 40, 20, 120, 80, 70}
```

Exercise 3.22. ($\star\star$)
For this problem you are not allowed to use an `ArrayList` or any helper methods, e.g., `.contains`, or methods from the `Arrays` class. You *may* (and should!) use a `HashSet<Integer>` to keep track of previously-seen peaks.

Joe the mountain climber has come across a large mountain range. He wants to climb only the tallest mountains in the range. Design the `int[] peakFinder(int[] H)` method that returns an array H' of all the peaks in an `int[]` of mountain heights H. A peak p is defined as an element of h at index i such that $p[i-1] < p[i]$ and $p[i] > p[i+1]$. If $i = 0$ or $i = |H| - 1$, Joe will not climb $p[i]$. Joe doesn't want to climb a mountain of the same height more than once, so you should not add any peaks that have already been added to H'. We present some test cases below. Hint: reuse the linear search algorithm from problem 1, except switch the `String` parameter for integers.

```
peakFinder({9, 13, 7, 2, 8})                         => {13}
peakFinder({8, 7, 8, 7, 8, 7, 8, 7})                 => {8}
peakFinder({111, 27, 84, 31, 5, 9, 4, 3, 2, 1, 64}) => {84, 9}
peakFinder({})                                       => {}
peakFinder({1})                                      => {}
peakFinder({1, 2})                                   => {}
peakFinder({1, 2, 1})                                => {2}
peakFinder({1, 2, 3, 2, 1})                          => {3}
```

Exercise 3.23. ($\star$)
A professor gives their students extra credit on an exam if they can guess the average within five percent of the actual average. Design the `boolean isGivenExtraCredit(ArrayList<Double> D, double g)` method that, when given a list of scores D and a guess g, returns whether a student is given extra credit.

Exercise 3.24. (⋆)
A village has members where each has a partner. These members are grouped in pairs inside an `ArrayList<Integer>` where each pair of indices represents a relationship of the village. I.e., `A.get(`$2i$`)` and `A.get(`$2i+1$`)` are in a relationship. A couple is considered the wisest if they have the highest combined age. Design the `ArrayList<Integer> wisest(ArrayList<Integer> A)` method that, when given a list of (integer) ages A, return a new `ArrayList<Integer>` containing the ages of the wisest pair. If there is a tie, return the pair that has the highest age overall. The order is not significant. We present a few test cases below. You can assume that there will always be an even number of village members.

```
wisest({31, 42, 43, 35, 21, 27, 24, 44}) => {43, 35} or {35, 43}
wisest({47, 51, 52, 48, 33, 67, 45, 35}) => {33, 67} or {67, 33}
```

Exercise 3.25. (⋆)
Design the `int missingNumber(Set<Integer> s)` method that, when given a set of integers whose values range from $1..n$ with one missing, return the number that is missing.

Exercise 3.26. (⋆⋆)
Design the `ArrayList<String> tokenize(String s, char d)` method that, when given a string s and a `char` delimiter d, returns an `ArrayList` of tokens split at the delimiter. You must do this by hand; you **cannot** call any `String` methods (except `.length` and `.charAt`).

Exercise 3.27. (⋆⋆)
Design the `HashMap<String, Integer> wordCount(String s)` method that, when given a string s, counts the number of words in the list. Store the results in a `HashMap<String, Integer>`. Assume that s is not cleaned. That is, you should first remove all punctuation (periods, commas, exclamation points, question marks, semi-colons, dashes, hashes, ampersands, asterisks, and parentheses) from s, producing a new string s'. Then, split the string based on spaces (remember `tokenize`?), and produce a map of the words to their respective counts. Do not factor case into your total; i.e., `"fAcToR"` and `"factor"` count as the same word. The ordering of the returned map is not significant.

```
String s = "Hello world, the world is healthy, is
            it not? I certainly agree that the world
            is #1 and healthy."
wordCount(s) => [{"hello", 1}, {"world", 3}, {"the", 2}
                 {"is", 3}, {"healthy", 2}, {"it", 1},
                 {"i", 1}, {"certainly", 1} {"agree", 1}
                 {"that", 1}, {"1", 1}, {"and", 1}, {"not", 1}]
```

Exercise 3.28. (⋆⋆⋆)
Design the `LinkedHashSet<String>largeToSmall(HashMap<String, Integer> M)` method that, when given a `HashMap<String, Integer>` M, returns a `LinkedHashSet` of `String` values where the words are inserted into the set in order of decreasing count. Words that have the same count do not need to be inserted in any particular order. E.g., `"is"` may come before `"world"`. You cannot sort M. Hint: create an array of size c (where c is the highest word count) where each element is a `LinkedHashSet<String>`. For every element e in M, add e to the set at index $M[i]$, where i is the count of e in M. Then, append these sets in reverse order according to their index. We provide some pseudocode below.

```
largeToSmall(M):
  // |M| is M.size().
  c = Get Highest Count Word in M
  array = [0 .. c-1]
  for i in [0 .. c-1]:
    array[i] = new LinkedHashSet<String>();
  for every key k in M:
    i = M.get(k)
    array[i-1].add(k)
  // Append all linkedhashsets in reverse order.
  return array[c-1] appended to array[c-2] ... appended to array[0]
```

```
largeToSmall(wordCount(s)) => {"world", "is", "the", "healthy",
                               "hello", "it", "i", "certainly",
                               "agree", "that", "1", "and", "not"}
```

Exercise 3.29. (⋆)
Design the `LinkedList<Integer> addLast(LinkedList<Integer> l, int v)` method that, when given a `LinkedList<Integer>` l and an `int` v, returns a *new* `LinkedList<Integer>` with the same elements plus v added to the end of l.

Exercise 3.30. (⋆)
Design the `LinkedList<Integer> set(LinkedList<Integer> l, int v, int i)` method that, when given a `LinkedList<Integer>` l, an `int` v, and an index i, returns a *new* `LinkedList<Integer>` with the same elements, except that the element at index i is, instead, the value v. If i is less than zero or exceeds the length of l, return `null`.

Exercise 3.31. (⋆)
Design the `int[] toArray(LinkedList<Integer> l)` method that, when given a linked list of integers l, returns an array containing the values from l.

Exercise 3.32. (⋆)
Design the `LinkedList<Integer> reverse(LinkedList<Integer> l)` method that, when given a linked list of integers l, returns a *new* linked list containing the elements of l, but reversed.

Exercise 3.33. (⋆⋆)
Design the `HashSet<Integer> moreThanThree(int[] A)` method that, when given an `int[]` A, returns a new `HashSet<Integer>` of values containing those values from A that occur strictly more than three times.

Exercise 3.34. (⋆⋆)
Design the `List<String> collectComments(String s)` method that, when given a string representing a (valid) Java program, returns an list containing all comments from the input program string. You cannot use any `String` helper methods (e.g., `strip`, `split`) to solve this problem nor can you use regular expressions.

Exercise 3.35. (⋆⋆)
Design the `List<String> removeSideBySideDups(List<String> ls)` that, when given a list of strings, returns a new list where all side-by-side duplicates are removed.

Exercise 3.36. (⋆⋆)
Design the `double postfixEvaluator(List<String> l)` method that, when given a list of binary operators and numeric operands represents as strings, returns the result of evaluating the postfix-notation expression. You will need to write a few helper methods

to solve this problem, and it is best to break it down into steps. First, write a method that determines if a given string is one of the four binary operators: "+", "-", "*", or "/". You may assume that any inputs that are not binary operators are operands, i.e., numbers. Then, write a method that applies a given binary operator to a list of operands, i.e., an ArrayList<Double>.

```
postfixEvaluator({"5", "2", "*", "5", "+"}) => 17
postfixEvaluator({"1", "2", "3", "4", "+"}) => 10
```

Exercise 3.37. (⋆⋆⋆)
Design the List<List<String>> displayOrders(List<List<String>> orderInfo) method that, when given an array of orders, returns a series of order specifications by customer table.

To be more specific, an order is a List<String> whose first element is the customer name, whose second element is the table number, and whose third element is the food that the customer is ordering.

Return the data as a list of rows of information. The first row displays the table headers. The first table header should be "Table", followed by the food in alphabetical order. Each successive row represents a table in increasing numerical order. Below is an input and output example.

```
{{"John", "2", "Chicken"}, {"Samantha", "3", "Pasta"},
 {"Tim", "2", "BBQ Chicken"}, {"Christina", "8", "Grilled Cheese"},
 {"Tymberlyn", "2", "Chicken"}, {"TJ", "3", "Water"}}
=>
{{"Table", "BBQ Chicken", "Chicken", "Grilled Cheese", "Pasta", "Water"},
 {"2", "1", "2", "0", "0", "0"},
 {"3", "0", "0", "0", "1", "1"},
 {"8", "0", "0", "1", "0", "0"}}
```

Exercise 3.38. (⋆)
Design the substitute method that, when given a String *exp* and an environment *env* HashMap<String, Integer>, substitutes each occurrence of an "identifier" for its value counterpart from the map.

```
substitute("f(x) = 3 * a + b", {<"a" : 10>, <"b" : 13>})
          => "f(x) = 3 * 10 + 13"
substitute("g(h, f(x)) = y + x", {<"q" : 200>})
          => "g(h, f(x)) = y + x"
```

Exercise 3.39. (⋆⋆⋆)
Design the unifiesAll method, which receives a HashMap<String, Integer> M of unification assignments and a list of goals ArrayList<LinkedList<String>> $\mathcal{G}$. Each goal $G \in \mathcal{G}$ is a tuple represented as a LinkedList; goals consist of two values x, y, and if it is possible for these to be the same value, then we say we can unify x with y. Any successful unifications that occur with variables not present in M should be added to M. We present some examples below (assume all values are strings; we omit the quotes out of conciseness). Hint: you might want to write an isVar predicate, which determines if a value is a variable or not, i.e., a value that does not start with a number.

```
M1 = {<x : 5>, <y : 10>, <z : 15>, <w : 5>}
G1  = {{x, x}, {10, 10}, {z, 15}, {x, w}}
unifiesAll(M1, G1) => true
M2 = {<x : 5>, <y : 10>, <z : 15>, <w : 5>}
```

```
G2 = {{x, y}}
unifiesAll(M2, G2) => false
M3 = {<x : 5>, <y : 10>, <z : 15>, <w : 5>}
G3 = {{q, x}, {w, 5}, {q, 5}, {q, w}}
unifiesAll(M3, G3) => true
M4 = {<x : 5>, <y : 10>, <z : 15>, <w : 5>}
G4 = {{q, x}, {q, 10}}
unifiesAll(M4, G4) => false
```

Exercise 3.40. (★★★)
Two strings s_1 and s_2 are isomorphic if we can create a mapping from s_1 from s_2. For example, the strings `"DCBA"` and `"ZYXW"` are isomorphic because we can map D to Z, C to Y, and so forth. Another example is `"ABACAB"` and `"XYXZXY"` for similar reasons. A non-example is `"PROXY"` and `"ALPHA"`, because once we map `"A"` to `"P"`, we cannot create a map between `"A"` and `"Y"`. Design the `boolean isIsomorphic(String s1, String s2)` method, which determines whether or not two strings are isomorphic.

Exercise 3.41. (★★)
Anagrams are strings that are formed by rearranging the letters of another string. For example, `"plea"` is an anagram for `"leap"`, but we consider an alphabetized anagram to be the alphabetized arrangement of letters for an anagram. As an example, `"aelrst"` is the alphabetized anagram for `"alerts"`, `"alters"`, `"slater"`, and `"staler"`. Design the `static Map<String, List<String>> alphaAnagramGroups(List<String> ls)` method, which maps all alphabetized anagrams to the strings in *ls* using the above criteria.

```
ls = ["presorting", "plea", "introduces", "anger", "leap", "petals",
      "donate", "plates", "range", "reductions", "rediscount",
      "tapers", "pale", "atoned", "staple", "repast", "reportings"]
alphaAnagramGroups(ls)
  => {{"aegnr", ["anger", "range"]},
      {"aelp", ["plea", "leap", "pale"]},
      {"aelpst", ["petals", "plates", "staple"]},
      {"aeprst", ["tapers", "repast"]}
      {"cdeinorstu", ["introduces", "reductions", "rediscount"]},
      {"adenot", ["donate", "atoned"]},
      {"eginoprrst", ["presorting", "reportings"]}}
```

Exercise 3.42. (★★★)
A SLC (simplified lambda calculus) expression takes one of the two forms: *varList* or λ*var*.*expr*, where *var* is a lower-case letter from u to z, *varList* is a sequence of variables, and *expr* is either a *var* or another SLC expression. We provide some examples below.

$$\lambda x.\lambda y.xyz$$
$$\lambda x.x$$
$$\lambda y.yzxw$$
$$\lambda w.\lambda x.\lambda y.\lambda z.z$$

Your job is to determine which variables are bound, which are free, and which are neither.

A bound variable is a variable that is bound by a λ and occurs in its expression. For example, in the expression $\lambda x.x$, x is bound.

A free variable is a variable that is *not* bound by a λ but does occur in an expression. For example, in the expression $\lambda y.\lambda x.zwv$, z, w, and v are free variables.

A variable that is neither free nor bound is a variable that is bound by a λ but does not occur in its expression. For example, in the expression $\lambda x.\lambda y.yz$, x is neither free nor bound.

Write the `static HashMap<String, String> classifyVars(String expr)` method, which returns a map of variables to their classification. We provide some examples below. You may assume that the input is a valid SLC expression and that no variables shadow one another. Use the values `V`, `B`, and `N` to represent free, bound, and neither, respectively.

```
classifyVars("xyz")        => <"x" : "F">, <"y" : "F">, <"z" : "F">
classifyVars("λx.λy.xyz") => <"x" : "B">, <"y" : "B">, <"z" : "F">
classifyVars("λx.λy.x")   => <"x" : "B">, <"y" : "N">
```

Exercise 3.43. (⋆⋆)
The *substitution cipher* is a text cipher that encodes an alphabet string A (also called the *plain-text alphabet*) with a key string K (also called the *cipher-text alphabet*). The A string is defined as `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"`, and K is any permutation of A. We can encode a string s using K as a mapping from A. For example, if K is the string `"ZEBRASCDFGHIJKLMNOPQTUVWXY"` and s is `"WE MIGHT AS WELL SURRENDER!"`, the result of encoding s produces `"VN IDBCY JZ VNHH ZXRRNFMNR!"`

Design the `subtitutionCipher` method, which receives a plain-text alphabet string A, a cipher-text string K, and a string s to encode, `substitutionCipher` should return a string s' using the aforementioned substitution cipher algorithm. You must follow the "design recipe" laid out in class. That is, you must write the method purpose statement comment, tests, and the implementation.

Exercise 3.44. (⋆)
Using the Stream API, design the `int sumOdd(List<Integer> l)` that, when given a `List<Integer>` l, filters out even numbers and then calculates the sum of the remaining odd numbers.

Exercise 3.45. (⋆)
Using the Stream API, design the `String conjoin(List<String> l)` method that, when given a `List<String>` l, concatenates all the strings together into a single string, separated by a comma.

Exercise 3.46. (⋆)
Using the Stream API, design the `Map<String, Integer> groupLength(List<String> l)` that, when given a `List<String>` l, groups the words by their length and counts how many words are there for each length. This means that the return value should be a `Map<String, Integer>`. There are a couple of ways to do this, and any way that correctly utilizes the Stream API is fine.

Exercise 3.47. (⋆)
Using the Stream API, design the `List<String> addYRemoveYY(List<String> l)` that, when given a `List<String>` l, returns a list where each string has `"y"` added at its end, omitting any resulting strings that contain `"yy"` as a substring anywhere.

Exercise 3.48. (⋆)
Using the Stream API, design the `List<Integer> squareAddFiveOmit(List<Integer>`

l) that, when given a List<Integer> l, returns a list of those numbers squared and adds five to the result, omitting any of the resulting numbers that end in 5 or 6.

Exercise 3.49. (⋆)
Using the Stream API, design the List<Integer> removeDups(List<Integer> l) method that, when given a List<Integer> l, removes all duplicate integers. Return this result as a new list.

Exercise 3.50. (⋆)
Using the Stream API, design the List<String> removeLongerThan(List<String> l, int n) method that, when given a List<String> l, removes all strings that are longer than a given integer n. Return this result as a new list.

Exercise 3.51. (⋆)
Using the Stream API, design the List<Double> filterThenSquare(List<Double> l) method that, when given a List<Double> l, removes all odd values, and squares the remaining values. Return this result as a new list.

Exercise 3.52. (⋆)
Using the Stream API, design the double filterDoubleAvg(List<Integer> l) method that, when given a List<Integer> l, removes all non-prime numbers, doubles each remaining value, and computes the average of said values. Return this result as a double value.

Exercise 3.53. (⋆)
Using the Stream API, design the Optional<Integer> min(List<Integer> l) method that, when given a List<Integer> l, returns the minimum value in the list. Return this result as an Optional<Integer>. If there are no values in l, return Optional.empty().

Exercise 3.54. (⋆)
Design the generic <K, V> V lookup(Map<K, V> m) method that, when given an Map<K, V> m and a value k of type K, returns the corresponding value (of type V) associated with the key k. If the key does not exist, return null. You will need to use the .equals method.

Exercise 3.55. (⋆)
Design the generic <T> String stringifyList(List<T> l) method that, when given an list of values l, returns a String of comma-separated values where each value is an element of l, but converted into a String. You'll need to use the .toString method implementation of the generic type T.

Exercise 3.56. (⋆⋆)
Design the generic <T extends Comparable<T>> Optional<T> min(List<T> ls) method, which receives a list of comparable elements and returns the minimum element in the list. It should return an Optional value of type T, where T is the type of the list. If the list is empty, return an empty Optional. Remember that T must be a type that implements the Comparable interface.

Exercise 3.57. (⋆⋆)
Design the <T extends List<Integer>> boolean areParallelLists(T t, T u) method that, when given two types of lists t and u that store integer values, determines whether or not they are "parallel." In this context, Two integer lists are parallel if they differ by a single constant factor. For example, where $t = \{5, 10, 15, 20\}$ and $u = \{20, 40, 60, 80\}$, t and u are parallel because every element in t multiplied by four gets

us a parallel element in u. This factorization is bidirectional, meaning that t could be $\{100, 200, 300, 200\}$ and u could be $\{10, 20, 30, 20\}$.

Exercise 3.58. ($\star\star$)
Design the `<T extends Set<U>, U extends Comparable<U>> boolean areEqualSets(T t, T u)` method that, when given two types of sets t and u that store comparable elements, returns whether they are equal to one another. Two sets are equivalent if they are subsets of each other. You must traverse over the sets; you **cannot** use the built-in `.equals` method provided by the `Set` implementations. 2

4. Classes and Objects

4.1 Classes

From the first page, we have made prolific use of classes, but in this section we will finally venture into the inner workings of a class, and how to create our own.

Classes are blueprints for *objects*. When we create a class, we declare a new type of object. As we stated, we have repeatedly used classes, e.g., strings, arrays, `Scanner`, `Random`, as well as those classes from the Collections API. Until now, however, we viewed these as forms of abstraction, whose details were not important.

To create a class, we use the `class` keyword, followed by the name of the class. The name of the class should be capitalized, and should, in general, describe a noun. All Java files describe a class and must be named accordingly. We, of course, have seen this repeatedly before, but we omitted the details.

Classes can *inherit* methods from other classes, a relationship called the superclass/subclass hierarchy. For now we will only mention that the `Object` class is the "ultimate" superclass, in which all classes are implicit subclasses. The `Object` class, in particular, has three methods that we will override in almost every class that we write: `equals`, for comparing two classes for equality, `toString`, a means of "stringifying" an object, and a third: `hashCode`, the significance of which we will return to soon. In subsequent chapters we will dive more into inheritance and hierarchies.

Example 4.1. Let's create a class called `Point`, which stores two `int` values representing a Cartesian coordinate x and y. By "store," we mean to say that x and y are *instance variables* of the `Point` class, also sometimes called *attributes*, *fields*, or *members* (in Java, we conventionally use the "instance variables" term). Instance variables denote the values associated with an arbitrary *instance* of that object (an instance may also be defined as an entity). For example, if we have a `Point` object `p`, then `p` has two instance variables, `x` and `y`, which are the x and y coordinates of `p`. Then, if we declare another point `p2`, then `p2` will have its own instance variables `x` and `y`, which are independent of `p`'s instance variables. In almost all circumstances, instance variables of a class should be marked as `private`. Instance variables that are `private` denotes that direct access to their values is granted only within the class definition. Lastly, for the time being, instance variables are immutable and cannot change. Thus, every instance variable will use the `final` keyword in its declaration, alongside the `UPPER_CASE` naming convention.

Speaking of *access modifiers*, we should mention the four that Java provides, even though we make prolific use of only three:

- A class, variable, or method declared with the `public` modifier is accessible to/by any other class. Variables that are `public` should be used sparingly.
- A class, variable, or method declared with the `private` modifier is accessible only to/by the class in which it is declared.
- A class, variable, or method declared without an access modifier, also called the *default access modifier*, behaves similarly to `public`, only that it is accessible only to/by classes in the same package. Packages are a means of organizing classes into groups, similar to directories.

The fourth and final access modifier is `protected`, which is similar to the default access modifier, but allows subclasses to access the variable or method.[1] We will not use `protected` in this text, but it is worth mentioning. As a corollary of sorts, any time that `protected` *can* be used, there is almost certainly a better design alternative, whether that means marking the variable/method as `private` or `public`, we are of the opinion that `protected` has few benefits.

Listing 4.1—`Point` Class Skeleton

```
class Point {

  private final int X;
  private final int Y;
}
```

We now want a way to create an instance of a `Point`. We declare instances of objects using the `new` keyword followed by the class constructor. *Constructors* are special methods that are called when we wish to create a new object of that class. Our `Point` class constructor can receive parameters, which we can use to initialize the relevant x and y instance variables. So, let's declare the constructor for our `Point` class. Constructors, in general, should be `public`, as we will need to call them from outside the class definition. Constructors are also special in that they do not have a return type, although phrasing it in this way leads to some confusion, because constructors have implicit return types. By implicit, we mean that we do not directly specify the return type, but Java knows that constructors are special, whose return types are of the class itself. We create a constructor by specifying the class name alongside any desired parameters.

Listing 4.2—`Point` Class Constructor

```
class Point {

  private final int X;
  private final int Y;

  public Point(int x, int y) {
    this.X = x;
    this.Y = y;
```

[1] If you have not heard of inheritance/subclasses yet, do not worry, as we will cover this in the next chapter; we explain it here to describe the relevant difference between the access modifiers.

```
  }
}
```

Remember that the purpose of the constructor is to initialize the class instance variables. So, unless we wish to use distinct identifiers for referencing the parameters and instance variables, we need to use the `this` keyword.[1] The `this` keyword refers to the current object, and aids in distinguishing between instance variables and parameters. Inside the constructor, we assign the value of the parameter `x` to the instance variable `x`. Should we opt to not use `this` on the left-hand variable identifier, then the parameter `x` would shadow the instance variable `x`, meaning that writing `x = x` assigns the parameter to itself. At last, we can create a `Point` object by calling the constructor, but wait, we have no way of accessing/referring to the instance variables of the `Point` object! We need to create *accessor methods* to retrieve the values of the instance variables. Accessor methods are `public` and should do exactly one thing: return the respective instance variable.

Listing 4.3—Designing Accessor Methods

```java
public Point {

  private final int X;
  private final int Y;

  public Point(int x, int y) {
    this.X = x;
    this.Y = y;
  }

  public int getX() { return this.X; }

  public int getY() { return this.Y; }
}
```

This principle of hiding the implementation details of a class is called *encapsulation*. Encapsulation is a key principle of object-oriented programming, and is one of the primary reasons why object-oriented programming is so powerful. It can be dangerous to directly modify or access the fields of an object.[2]

Creating an instance of the `Point` class is identical to creating an instance of any other arbitrary class. Though, we should first explain a slight terminology distinction.

Declaring, or initializing, an object refers to typing the class name followed by the variable name. For instance, the following code declares a `Point` object `p`.

```java
Point p;
```

By default, p points to `null`, since we have not yet created an instance of the `Point` class. We can create an instance of the `Point` class by invoking its constructor, an action otherwise called *object instantiation*. We use the `new` keyword and pass the desired x and y coordinates.

```java
Point p = new Point(3, 4);
```

We should write some tests to ensure that our `Point` class is working as expected. We note that this may seem redundant for such a simple class and the fact that the accessor

[1] Some software engineers and projects use identifier prefixes to refer to instance variables.

[2] By "dangerous," we mean to suggest that it is prone to logic errors.

methods do nothing more than retrieve instance variable values, but it is a good habit for beginning object-oriented programmers.

Listing 4.4—Testing Accessor Methods

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

class PointTester {

  @Test
  void testPoint() {
    Point p = new Point(3, 4);
    assertEquals(3, p.getX());
    assertEquals(4, p.getY());
  }
}
```

Of course, testing the accessor methods is a little boring, so let's override the `toString` method to print a stringified representation of the `Point` class. Every object in Java has a `toString` method, which returns a string representation of the object. By default, the `toString` method returns the class name followed by the object's hashcode. We can override the `toString` method by declaring a `public` method with the signature `public String toString()`. We can then return a string representation of the object. In this case, we will return a string of the form `"(x=`x`, y=`y`)"`, where x and y refer to the respecitve instance variables.

Listing 4.5—Overriding the `toString` Method

```
class Point {
  // ... previous code not shown.

  @Override
  public String toString() {
    return String.format("(x=%d, y=%d)", this.X, this.Y);
  }
}
```

Testing the `toString` method provides more interesting results, since it requires us to not only override the default implementation of `toString`, but it also ensures that our constructor correctly initializes the instance variables. Because we will refer to p across several methods, we can declare it as an instance variable of our `PointTester` class so as to reduce the redundant object instantiation.

Listing 4.6

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

class PointTester {

  private final Point P = new Point(3, 4);

  @Test
  void testPointAccessors() {
    assertEquals(3, P.getX());
    assertEquals(4, P.getY());
```

```
  }

  @Test
  void testPointToString() {
    assertEquals("(x=3, y=4)", P.toString());
  }
}
```

In addition to the `toString` method, we might also write other methods associated with a `Point` object. For example, we might want to calculate the distance between two points. We can write a `public` method that takes a `Point` object as a parameter and returns the distance between the two points, the first being the *implicit parameter*, and the second being the *explicit parameter*. We say the first is *implicit* because, under the hood, all class methods take an implicit parameter, which is the object on which the method is called, which is accessible through `this`. We say the second is *explicit* because we explicitly pass the object as a parameter.

This is also a good time to bring up another terminology distinction. Some programming languages use *functions*, others use *procedures*, *subroutines*, or *methods*. Going from simplest to most complex, subroutines are simply a sequence of instructions that are executed in order. Procedures are subroutines that return a value. Functions are procedures that take parameters. Methods are functions that are associated with a class. In Java, we use the term *method* to refer to all of these, since all methods must be associated with a class. A language like C++, on the other hand, distinguishes between the two: *functions* refer to subroutines, procedures, or parameter-receiving procedures that are not associated with a class; *methods* are subroutines, procedures, or functions embedded inside a class definition.

Returning to the `Point` class, we will now write `distance`, which receives a `Point` as a parameter and returns the Euclidean distance from `this` to the parameter. Before doing so, however, we should write a few tests. Conveniently, the three points that we test all have a distance of five between each other.

Listing 4.7—Testing the `distance` Method

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

class PointTester {

  private static final double DELTA = 0.01;

  private final Point P1 = new Point(3, 4);
  private final Point P2 = new Point(6, 8);
  private final Point P3 = new Point(0, 0);

  @Test
  void testPointDistance() {
    assertAll(
      () -> assertEquals(5, P1.distance(P2), DELTA),
      () -> assertEquals(5, p2.distance(P1), DELTA),
      () -> assertEquals(5, P1.distance(P3), DELTA),
      () -> assertEquals(5, P3.distance(P1), DELTA),
      () -> assertEquals(5, P2.distance(P3), DELTA),
      () -> assertEquals(5, P3.distance(P2), DELTA));
  }
}
```

Listing 4.8—Distance Between Two Points

```
class Point {
  // ...  previous code not shown.

  /**
   * Determines the Euclidean distance between two points.
   * @param p the other point.
   * @return the distance between this point and p.
   */
  public double distance(Point p) {
    int dx = this.X - p.X;
    int dy = this.Y - p.Y;
    return Math.sqrt(dx * dx + dy * dy);
  }
}
```

The `distance` method is called an *instance method* because it is associated with an instance of the class. We can also write *static methods*, which are not associated with an object, but rather the class as a whole. Static methods are declared using the `static` keyword. Static methods are useful for utility methods that are not associated with a particular instance of the class. All methods designed up until this chapter were static methods, which were not associated with the class in which they resided.

Method Overloading

A method is identified by two attributes: its name, and its signature. Java allows us to *overload* a method or constructor by using the same identifier but different parameters.

Example 4.2. Let's overload the `distance` method by writing a version that does not receive a parameter at all, and instead returns the magnitude/distance from the point to the origin. Fortunately this is extremely easy, because we can make use of the existing `distance` method that does receive a `Point`; we can pass it the origin point, namely `new Point(0, 0)`, and everything works out wonderfully! Because this version of `distance` simply refers to the existing definition, which we have thoroughly tested, we will omit a separate tester.

Listing 4.9—Distance Method Overloading

```
class Point {
  // Previous code not shown.

  /**
   * Computes the distance from this point to the origin,
   * i.e., (0, 0).
   * @return returns the magnitude of this distance.
   */
   public double distance() {
     return this.distance(new Point(0, 0));
   }
}
```

We could, if desired, overload the `Point` constructor as well. Though, it makes little sense to do so in this specific instance, since a point is defined by its two coordinate members. In subsequent sections, however, we will overload the constructor and demonstrate its utility/practicality.

Example 4.3. Let's create the static method `random`, which returns a random `Point` object. We will use the `Random` class to generate a random radius and angle as a polar coordinate. Then, we will convert the polar coordinate to Cartesian coordinates. Let's also add a parameter that specifies the maximum radius.

Because the `random` method generates a random point, we cannot reasonably write a test that asserts the exact location of the point without prior knowledge of the random seed. Instead, we can write a test that asserts that the point is within a certain radius of the origin.

Listing 4.10—Testing the `random` Method

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

class PointTester {

  private static final double DELTA = 0.01;

  @Test
  void testPointRandom() {
    assertTrue(Point.random(10).distance() <= 10);
    assertTrue(Point.random(1).distance() <= 1);
    assertTrue(Point.random(5).distance() <= 5);
    assertTrue(Point.random(5000000).distance() <= 5000000);
  }
}
```

Listing 4.11—Random Point Generation Method

```
import java.util.Random;

class Point {

  /**
   * Generates a random point with a maximum radius.
   * @param maxRadius the maximum radius.
   * @return a random point.
   */
  public static Point random(double maxRadius) {
    Random r = new Random();
    double radius = r.nextDouble(maxRadius);
    double angle = r.nextDouble() * Math.PI * 2;
    int x = (int) (radius * Math.cos(angle));
    int y = (int) (radius * Math.sin(angle));
    return new Point(x, y);
  }
}
```

We have seen static methods, but what about static variables? A static variable, as we mentioned before, is a variable that is associated with the class and not a specific instance thereof. We can declare a static variable using the `static` keyword. Static variables are useful for storing information that is shared across all instances of the class. Static variables may be either `private` or `public`, depending on whether we want to allow direct access to the variable. As with instance variables, however, be wary of granting direct access, since it may lead to logic errors.

Example 4.4. Suppose we want to keep track of how many instances of `Point` have been instantiated. Since this is a property of the `Point` class rather than an instance of the class, we can declare a static variable `count` to keep track of the number of instances, which is incremented inside the constructor. To remain consistent with our recurring theme of encapsulation, `count` will be declared as `private`, and we will write a static method `getCount` to retrieve the number of instances, which is invoked on the class.[1] When testing, we need to be careful to only instantiate instances of `Point` when we are ready to check the status of `count`, since the static `count` variable is incremented inside the constructor.

Listing 4.12—Testing the `count` Static Variable

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

class PointTester {

  @Test
  void testPointCount() {
    assertEquals(0, Point.getCount())
    Point p1 = new Point(3, 4);
    assertEquals(1, Point.getCount());
    Point p2 = new Point(6, 8);
    assertEquals(2, Point.getCount());
    Point p3 = new Point(0, 0);
    assertEquals(3, Point.getCount());

    // Even though we lose reference to p, the static varible still increments!
    for (int i = 0; i < 10; i++) {
      Point p = new Point();
    }
    assertEquals(13, Point.getCount());
  }
}
```

Listing 4.13—`Point` Class with `count` Static Variable

```
class Point {

  private static int count = 0;

  private final int X;
  private final int Y;

  public Point(int x, int y) { this.X = x; this.Y = y; count++; }

  public static int getCount() { return count; }
}
```

Notice that, inside the `getCount` method, we do not refer to `count` with `this`, since `count` is a static variable and not an instance variable. Prefixing the `count` variable with `this` results in a compiler error.

Example 4.5. Imagine we want to store a collection of `Point` objects in a data structure such as a `HashSet`. The question that arises from this decision is apparent: how do we de-

[1] It is possible to invoke a static method on an instance of the class, but it is considered bad practice and unnecessary.

termine if a `Point` is already inside the set? We need to override two important methods from `Object`: `equals` and `hashCode`. The `equals` method of an object determines how we wish to compare two instances of the class. In this circumstance, let's say that two points are equal according to `equals` if and only if they share the same x and y coordinates. Overriding the `equals` method from the `Object` class requires correctly copying the signature, the sole parameter being an `Object` that we want to check for type equality. In other words, we first want to verify that the passed object to the `equals` method is, in fact, a `Point`, otherwise they cannot possibly be equal. We can use the `instanceof` operator to our advantage. From here, if the input parameter is a `Point`, we can cast the parameter to a `Point`, then check whether or not the coordinates match.

Listing 4.14—Testing Equality Over Points

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

class PointTester {

  @Test
  void testEquals() {
    assertAll(
      () -> assertEquals(new Point(3, 3), new Point(3, 3)),
      () -> assertNotEquals(new Point(3, 4), new Point(3, 7)),
      () -> assertNotEquals(new Point(7, 4), new Point(10, 4)),
      () -> assertNotEquals(new Point(10, 30), new Point(3, 7)));
  }
}
```

Listing 4.15—Overriding the Equals Method in Point

```java
class Point {

  // ...  other methods not shown.

  @Override
  public boolean equals(Object obj) {
    if (!(obj instanceof Point)) { return false; }
    else {
      Point othPt = (Point) obj;
      return this.x == othPt.x && this.y == othPt.y;
    }
  }
}
```

Let's try to create an instance of a `HashSet`, then iterate over the elements of the set after adding two of the same `Point` instances, i.e., points that share coordinates. Doing so demonstrates a glaring flaw: the set appears to have added both `Point` instances to the set despite their sharing of coordinates! The reason is incredibly subtle and easy to miss: the `Object` class invariant states that, if two objects are equal according to `equals`, then their hashcodes must also be equal. The hashcode of an object is simply an integer used for quick access/lookup in hashable data structures such as `HashSet` and `HashMap`. Indeed, the problem is that we forgot to override `hashCode` after overriding the `equals` method. Bloch [Bloch, 2018] states, as a principle, that whenever we override `equals`, we should accompanyingly override the `hashCode` implementation. Now, you might wonder: "How can I hash an object?" Fortunately Java has a method in the `Objects` class called `hash`, which receives any number of arguments and runs them through a hashing algorithm, thereby returning the hash of the parameters. When overriding `hashCode` we should include all instance variables of the object to designate that all of the properties affect the object's hashcode. After fixing the issue, we see that our `HashSet` now correctly contains only one of the `Point` objects that we add.

Listing 4.16—Testing Hashcode Implementation for Point

```java
import static Assertions.assertTrue;

import java.util.Set;
import java.util.HashSet;

class PointTester {

  @Test
  void testHashSetPoint() {
    Set<Point> p = new HashSet<>();
    p.add(new Point(3, 3));
    p.add(new Point(3, 3));
    assertTrue(p.size() == 1);
  }
}
```

Listing 4.17—Adding Hashcode Implementation for Point

```java
import java.util.Objects;

class Point {

  // ...  other methods not shown.

  @Override
  public int hashCode() {
    return Objects.hash(this.x, this.y);
  }
}
```

Example 4.6. Let's amplify the complexity a bit by designing a "21" card game, which is a card game where the players try to get a card value total of 21 without going over. We should think about the design process of this game, namely what classes we need to create. It makes sense to start off with a `Card` class, which stores its suit and its numeric value. Because a suit is one of four possibilities, each of which uses a different symbol, we can create another class called `Suit`. In `Suit` we instantiate four static instances of `Suit`, each of which represents one of the four valid suits. Its constructor is privatized because we, as the programmers, define the four possible suits; it should not be possible for the user to define their own custom suit, at least for this particular game. The notion of `Suit` being an instance variable of `Card`, and only exists due to `Card` is called *object composition*. Lastly, we will provide a method that returns an `Iterator<Suit>` over the four suit possibilities to make our lives easier when designing the `Deck` class. The method should be static so it is accessible through the class.

Listing 4.18—Suit Class

```
class Suit {

  public static final Suit CLUBS = new Suit("♣");
  public static final Suit DIAMONDS = new Suit("♢");
  public static final Suit HEARTS = new Suit("♡");
  public static final Suit SPADES = new Suit("♠");
  public static final int NUM_SUITS = 4;

  private final String S_VAL;

  private Suit(String s) { this.S_VAL = s; }

  public static Iterator<Suit> iterator() {
    return new ArrayList<Suit>(List.of(CLUBS, DIAMONDS, HEARTS, SPADES))
                               .iterator();
  }

  @Override
  public String toString() { return this.S_VAL; }
}
```

Testing the `Card` class is simple and straightforward; we only need to test one method, the `toString` method, since testing `getValue`, at this point, is superfluous. We could also test the `Suit` class, but we will not do so here, given that the only useful methods are to retrieve the instance variables and the iterator.

Listing 4.19—Testing the Card Class

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

class CardTester {

  @Test
  void testCardToString() {
    assertAll(
      () -> assertEquals("2 of ♣", new Card(Suit.CLUBS, 2).toString()),
      () -> assertEquals("3 of ♢", new Card(Suit.DIAMONDS, 3).toString()),
      () -> assertEquals("4 of ♡", new Card(Suit.HEARTS, 4).toString()),
      () -> assertEquals("5 of ♠", new Card(Suit.SPADES, 5).toString()));
  }
}
```

Listing 4.20—Card Class

```
class Card {

  private final Suit SUIT;
  private final int VAL;

  public Card(Suit suit, int value) { this.SUIT = suit; this.VAL = value; }

  @Override
  public String toString() { return String.format("%d of %s", this.VAL, this.SUIT)
      ; }

  public int getValue() { return this.VAL; }
```

```
}
```

In a standard fifty-two deck of cards, some are "special," e.g., the Jacks, Queens, Kings, and Ace cards. To simplify our design, these cards will be treated the same as a "ten" card, showing no syntactic nor semantic difference. Now that we have a class to represents cards, let's design the `Deck` class, which stores an `ArrayList<Card>` representing the current state of the deck. It also contains a static variable representing the maximum number of allowed cards. For our purposes, as we alluded to, this quantity is fifty-two. In the `Deck` constructor, we will call the `populateDeck` method, which adds four cards of the same value, but of each suit. So, to exemplify, this means that there are four cards whose value is three, where each is one of the four suits. We make use of the iterator from the `Suit` class to simplify our deck population. Only the `Deck` class needs to know how to populate an initial (empty) deck, so we privatize its implementation.

To test a `Deck`, we can write a `drawCard` method, which retrieves the "top-most" card on the deck to see if it is in the correct order. According to our implementation of the iterator, the top-most cards should have values of ten and be of the same suit. From there, we can draw three more cards to ensure they are of values nine, eight, and seven of the same suit. The iterator places `DIAMOND` as the final suit, so this is what we will assume in our tester. It might also be beneficial to test the `isEmpty` method, which returns `true` if the deck is empty, and `false` otherwise. We can test this functionality by drawing all fifty-two cards from the deck and ensuring that the deck is empty afterwards. Note that we draw four tens because there are no "Kings," "Queens," or "Jacks" in the deck.

Listing 4.21—Testing the Deck Class

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

class DeckTester {

  @Test
  void testDeckDrawCard() {
    Deck d = new Deck();
    assertAll(
      () -> assertEquals("10 of ◇", d.drawCard().toString()),
      () -> assertEquals("10 of ◇", d.drawCard().toString()),
      () -> assertEquals("10 of ◇", d.drawCard().toString()),
      () -> assertEquals("10 of ◇", d.drawCard().toString()),
      () -> assertEquals("9 of ◇", d.drawCard().toString()));
  }

  @Test
  void testDeckIsEmpty() {
    Deck d = new Deck();
    for (int i = 0; i < 52; i++) {
      assertFalse(d.isEmpty());
      d.drawCard();
    }
    assertTrue(d.isEmpty());
  }
}
```

Listing 4.22—Deck Class

```
import java.util.ArrayList;
import java.util.Iterator;

class Deck {

  private static final int MAX_NUM_CARDS = 52;

  private final ArrayList<Card> CARDS;

  public Deck() {
    this.CARDS = new ArrayList<Card>();
    this.populateDeck();
  }

 /**
  * Retrieves a card from the "top" of the deck. If the
  * deck is empty, we return null.
  * @return the top-most card in the deck, or null if the deck is empty.
  */
  public Card drawCard() {
    return this.CARDS.isEmpty() ? null : this.CARDS.remove(this.CARDS.size() - 1);
  }

  /**
   * Determines if the deck is empty.
   * @return true if the deck contains no cards, and false otherwise.
   */
  public boolean isEmpty() {
    return this.CARDS.isEmpty();
  }

  /**
   * Instantiates the deck to contain all 52 cards.
   * Note that the deck contains cards in-order by suit. There are
   * no face cards in the deck, i.e., no Jack, Queen, King, nor ace.
   * All cards have a value between 1 and 10.
   */
  private void populateDeck() {
    // For every suit, create 13 cards, the last four of which all have
    // a value of ten.
    Iterator<Suit> it = Suit.iterator();
    while (it.hasNext()) {
      Suit s = it.next();
      for (int i = 1; i <= MAX_NUM_CARDS / Suit.NUM_SUITS; i++) {
        Card c = new Card(s, Math.min(10, i));
        this.CARDS.add(c);
      }
    }
  }
}
```

Hopefully the `populateDeck` method is intuitive and not intimidating. All we do is create fifty two cards, thirteen of which are of the same suit, and add them to the deck. We use the `Math.min` method to ensure that the value of the card is at most ten, since we do not have “King,” “Queen,” or “Jack” cards. We also use the ternary operator to check if the deck is empty before drawing a card. If the deck is empty, we return `null`.

Finally we come to the `Player` class, which stores a "hand" containing the cards in their possession. Fortunately this is a very straightforward definition and contains four one-line methods: `addCard`, `clearHand`, `getScore`, and `toString`. The former two are trivial to explain, as is `toString`, whereas `getScore` is the only slightly convoluted method. The idea is to return an integer that represents the total value of the cards in the player's hand. Since streams were introduced a couple of chapters ago, we will once again use them to our advantage.

Listing 4.23—`Player` Class

```
import java.util.ArrayList;

class Player {

  private final ArrayList<Card> HAND;

  public Player() { this.HAND = new ArrayList<Card>(); }

  /**
   * Adds a card to the player's hand.
   * @param c card to add to the player's hand.
   */
  public void addCard(Card c) { this.HAND.add(c); }

  /**
   * Removes all cards from the player's hand.
   */
  public void clearHand() { this.HAND.clear(); }

  /**
   * Determines the player's score.
   * @return the player's score.
   */
  public int getScore() {
    return this.HAND.stream()
                    .map(c -> c.getValue())
                    .reduce(0, Integer::sum);
  }

  @Override
  public String toString() {
    return String.format("Score: %d\nHand: %s\n",
                         this.getScore(), this.HAND.toString());
  }
}
```

Using the capabilities of `Player`, `Deck`, and `Card`, we will write `TwentyOne`: the class that runs a game of "twenty-one." The game logic is simple: if the game is still running, clear the player's hand, create a new deck of cards, shuffle them, and give the player two. Then, ask the player if they want to draw another card. If they do, draw a card and add it to their hand. If they do not, then the game is over. If the player's score is greater than twenty-one, then the player loses. Otherwise, the player wins. We will also write a `main` method that runs the game. We will not write any tests for this class, since it interacts with the user through the `Scanner` class.

It should be noted that this version of "twenty-one" only has the objective of getting as close as possible to a hand containing cards with a value that sums to twenty one, compared to a more-traditional card game where multiple players exist, with a dealer to distribute cards. As exercises, there are many ways to enhance the game, including adding a “high score” board to keep track of previous game outcomes, introducing CPU players to automatically poll cards from the deck to beat the main player, or even adding more human players through standard input/output iteractions.

Listing 4.24—TwentyOne Class

```java
import java.util.Scanner;

class TwentyOne {

  private static final int MAX_SCORE = 21;

  private final Player PLAYER;

  public TwentyOne() { this.PLAYER = new Player(); }

  /**
   * Plays a game of "21", where the player has to draw cards until they
   * get as close to 21 as possible without going over.
   */
  public void playGame() {
    Scanner in = new Scanner(System.in);
    boolean continuePlaying = true;
    while (continuePlaying) {
      // Clear the player's hand.
      this.player.clearHand();

      // Create and shuffle the deck.
      Deck d = new Deck();
      d.shuffleDeck();

      // First, deal two cards.
      this.PLAYER.addCard(d.drawCard());
      this.PLAYER.addCard(d.drawCard());

      // While the player has not "busted", ask them to draw a card or stand.
      while (this.PLAYER.getScore() <= this.MAX_SCORE) {
        System.out.println(this.PLAYER);
        System.out.println("Do you want to draw? (Y/n)");
        String resp = in.nextLine();
        if (resp.equals("Y")) { this.PLAYER.addCard(d.drawCard()); }
        else { break; }
      }

      // Print the final results of the player.
      System.out.println(this.PLAYER);
      if (this.PLAYER.getScore() > this.MAX_SCORE) {
        System.out.println("You lose!");
      } else {
        System.out.println("You did not go over!");
      }

      // Determine if we're still playing.
      System.out.println("Do you want to continue playing?");
      String resp = in.nextLine();
      continuePlaying = resp.equals("Y");
    }
```

```
  }

  public static void main(String[] args) {
    new TwentyOne().playGame();
  }
}
```

Designing interactive games is a great exercise in object-oriented programming, as well as the culmination of other discussed topics.

Example 4.7. Let's design the `Rational` class, which stores a rational number as a numerator and denominator. In doing so we will create methods for adding, subtracting, multiplying, and dividing rational numbers. Testing is paramount with this example, so we will be sure to write plenty. Recall the definition of a rational number: a number that can be expressed as the ratio of two integers p and q, namely p/q. We are acutely familiar with how to perform basic operations on fractions from grade school, so we will glide through the actual mathematics and focus more on the Java implementation and class design.

The `Rational` constructor receives two integers p and q, and assigns them as instance variables. The `toString` method is trivial to write and only involves us adding a slash between our numerator and denominator. Though, let's back up for a second and rethink the constructor. Do we really want to be able to store fractions that are not in their simplest form? For example, do we want to allow the user to create a `Rational` object with a numerator of 2 and a denominator of 4? The answer is probably not, meaning that we should add a method that simplifies the fraction. We can do this by finding the greatest common divisor of the numerator and denominator, and dividing both by that value. Euclid's algorithm for finding the greatest common divisor of two integers works wonderfully here. Due to its trivial implementation and the fact that it is a tail recursive algorithm exercise from the previous chapters, we will omit its implementation.

Listing 4.25—Testing the `Rational` Class

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

class RationalTester {

  @Test
  void testRationalToString() {
    assertAll(
      () -> assertEquals("1/2", new Rational(1, 2).toString()),
      () -> assertEquals("3/400", new Rational(3, 400).toString()),
      () -> assertEquals("1/1305", new Rational(5, 6525).toString()),
      () -> assertEquals("3591/46562", new Rational(7182, 93124).toString()),
      () -> assertEquals("7/32", new Rational(7, 32).toString()),
      () -> assertEquals("9388/48122", new Rational("4694/24061").toString()),
      () -> assertEquals("1/1", new Rational(1, 1).toString()));
  }
}
```

Listing 4.26—`Rational` Class

```
class Rational {

  private final long NUMERATOR;
  private final long DENOMINATOR;
```

```java
  public Rational(long numerator, long denominator) {
    long gcd = gcd(numerator, denominator);
    this.NUMERATOR = numerator / gcd;
    this.DENOMINATOR = denominator / gcd;
  }

  @Override
  public String toString() {
    return String.format("%ld/%ld", this.NUMERATOR, this.DENOMINATOR);
  }
}
```

To add two rational numbers r_1 and r_2, they must share a denominator. If they do not, then we must find a common denominator by multiplying the denominators together, then multiplying the relevant numerators by the reciprocals of the denominator. For instance, if we want to add $2/3$ and $7/9$, the (not-necessarily lowest) common denominator is $3 \cdot 9 = 27$. We then multiply 2 by 9 and 7 by 3 to get $18/27$ and $24/27$. Adding across the numerators results in $42/27$, which reduces to $14/9$. Since we wish to preserve the original rational number, we will write a method that returns a new `Rational` rather than modifying the one we have in-place (this also allows us to omit a step in which we simplify the resulting fraction, since the constructor takes care of this task).

Listing 4.27—Testing Addition of Rationals

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

class RationalTester {

  @Test
  void testRationalAdd() {
    assertAll (
      () -> assertEquals("14/9",
                      new Rational(2, 3).add(new Rational(7, 9)).toString()),
      () -> assertEquals("5/6",
                      new Rational(1, 2).add(new Rational(1, 3)).toString()),
      () -> assertEquals("1/3",
                      new Rational(1, 4).add(new Rational(1, 12)).toString()),
      () -> assertEquals("1/4",
                      new Rational(1, 8).add(new Rational(1, 8)).toString()),
      () -> assertEquals("1/8",
                      new Rational(1, 16).add(new Rational(1, 16)).toString()),
      () -> assertEquals("1/16",
                      new Rational(1, 32).add(new Rational(1, 32)).toString()),
      () -> assertEquals("2/1",
                      new Rational(32, 32).add(new Rational(32, 32)).toString()
                           ));
  }
}
```

Listing 4.28—Adding Two Rational Numbers

```java
class Rational {
  // Other details not shown.

  /**
   * Adds two rational numbers.
   * @param r the other rational number.
```

```
   * @return the (simplified) sum of this and r.
   */
  public Rational add(Rational r) {
    long commonDenominator = this.DENOMINATOR * r.DENOMINATOR;
    long newNumerator = this.NUMERATOR * r.DENOMINATOR
                      + r.NUMERATOR * this.DENOMINATOR;
    return new Rational(newNumerator, commonDenominator);
  }
}
```

Due to the correspondence to addition, we will leave `subtraction` as an exercise to the reader. We can now do multiplication, which is even simpler than addition; all that is needed is to multiply the numerators and denominators together. We will also leave `division` as an exercise to the reader. We also encourage the reader to write methods for comparing rationals for equality, as well as greater than/less than. Plus, we could extend this system to support `BigInteger` values for the numerator and denominator, which would allow us to represent arbitrarily large rational numbers. This, in turn, would require updating all methods to use `BigInteger` arithmetic, which is a good exercise in itself.

Listing 4.29—Testing Multiplication of Rationals

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

class RationalTester {

  @Test
  void testRationalMultiply() {
    assertAll (
      () -> assertEquals("14/27",
                        new Rational(2, 3).multiply(new Rational(7, 9)).toString
                             ()),
      () -> assertEquals("1/6",
                        new Rational(1, 2).multiply(new Rational(1, 3)).toString
                             ()),
      () -> assertEquals("1/48",
                        new Rational(1, 4).multiply(new Rational(1, 12)).toString
                             ()),
      () -> assertEquals("1/64",
                        new Rational(1, 8).multiply(new Rational(1, 8)).toString
                             ()),
      () -> assertEquals("1/25",
                        new Rational(1, 5).multiply(new Rational(1, 5)).toString
                             ()),
      () -> assertEquals("1/1",
                        new Rational(1, 1).multiply(new Rational(1, 1)).toString
                             ()));
  }
}
```

Listing 4.30—Multiplying Two Rational Numbers

```
class Rational {
  // Other details not shown.

  /**
   * Multiplies two rational numbers.
   * @param r the other rational number.
```

```
   * @return the (simplified) product of this and r.
   */
  public Rational multiply(Rational r) {
    return new Rational(this.NUMERATOR * r.NUMERATOR,
                        this.DENOMINATOR * r.DENOMINATOR);
  }
}
```

Example 4.8. Let's now use classes to demonstrate a theoretically powerful idea: translating standard recursive methods into ones that use iteration. We have seen how to translate a tail recursive method, but standard recursion was left out of the discussion. In general, any recursive method can be rewritten to use iteration. The problem we encounter with standard recursive algorithms is that they often blow up the procedure call stack, of which is very limited in size for most programming languages. What if we did not push anything to the call stack at all, and instead implement our own stack? In doing so, we delegate the space requirements of the recursive calls from the (call) stack to the heap, of which there is orders of magnitude more space. This solution is neither fast nor space-efficient, but serves to show that naturally standard recursive algorithms, e.g., factorial, can still use standard recursion in a sense.

To create our own stack, we first need to decide what goes inside the stack. We know that each method call pushes an activation record, or a stack frame, to the procedure call stack containing the existing local variables and parameters. For the sake of simplicity, let's assume that our methods never declare local variables. We need a class that stores variable identifiers to values, which can be any type. A simple solution to the "any type" problem is to use the `Object` class that all classes implicitly extend. So, the `StackFrame` class stores a `Map<String, Object>` as an instance variable. Its constructor receives no arguments because we do not know a priori how many parameters any arbitrary user of `StackFrame` will require. To compensate, let's write the `addParam` method that receives a `String` and an `Object`, enters those into the existing map, and returns the existing instance.[1] We design the method in this fashion to prevent the need to separately instantiate the frame, then add its parameters on separate lines, which would be required if `addParam` were of type void.

Listing 4.31—Stack Frame Definition

```
import java.util.HashMap;
import java.util.Map;

class StackFrame {

  private Map<String, Object> PARAMS;

  public StackFrame() { this.PARAMS = new HashMap<>(); }

  public Object getParam(String s) { return this.PARAMS.get(s); }

  public StackFrame addParam(String s, Object o) {
    this.PARAMS.put(s, o);
    return this;
  }
}
```

[1]This idea resembles the *builder* design pattern, which we will discuss in Chapter 9.

Now, let's translate the standard recursive `fact` method, which will receive a `BigInteger`, and return its factorial. Below we show the recursive version. From this, we write the `factLoop` method that instantiates a `Stack<StackFrame>` to replicate the call stack. We begin the process by pushing the initial frame, which stores the initial input argument. This is followed by a variable to keep track of the "return value," which should match the type of the standard recursive method (for our purposes of factorial, this is a `BigInteger`). Our loop continues as long as there is a stack frame to pop, and the core logic of the algorithm, namely $n!$, is identical to our standard recursive counterpart.

Listing 4.32—Testing Two Factorial Implementations

```java
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static StackFrameDriver.fact;
import static StackFrameDriver.factLoop;

import java.util.BigInteger;

class StackFrameTester {

  @Test
  void testFact() {
    assertAll (
      () -> assertEquals(new BigInteger("1"),
                         fact(new BigInteger("0"))),
      () -> assertEquals(new BigInteger("120"),
                         fact(new BigInteger("5"))),
      () -> assertEquals(new BigInteger("3628800"),
                         fact(new BigInteger("100"))),
    );
  }

  @Test
  void testFactLoop() {
      assertAll (
      () -> assertEquals(new BigInteger("1"),
                         factLoop(new BigInteger("0"))),
      () -> assertEquals(new BigInteger("120"),
                         factLoop(new BigInteger("5"))),
      () -> assertEquals(new BigInteger("3628800"),
                         factLoop(new BigInteger("100"))),
    );
  }
}
```

Listing 4.33—Factorial Method Definitions

```java
import java.util.Stack;
import java.util.BigInteger;

class StackFrameDriver {

  public static BigInteger fact(BigInteger n) {
    if (n.compareTo(BigInteger.ONE) <= 0) {
      return n.add(BigInteger.ONE);
    } else {
      return n.multiply(fact(n.subtract(BigInteger.ONE)));
    }
```

```
  }

  public static BigInteger factLoop(BigInteger n) {
    Stack<StackFrame> sf = new Stack<>();
    BigInteger res = n;
    sf.push(new StackFrame().addParam("n", n));
    while (!sf.isEmpty()) { /* TODO. */ }
    return res;
  }
}
```

Turning our attention to the innards of the loop, we must accurately replicate the procedure call stack actions. Thus, we first pop the existing frame, extract the desired parameters to work with from its map, then perform the algorithm's logic.

Listing 4.34—Finishing "Loop" Factorial Definition

```
class StackFrameDriver {

  // ...  other implementation not shown.

  public static BigInteger factLoop(BigInteger n) {
    Stack<StackFrame> sf = new Stack<>();
    BigInteger res = BigInteger.ONE;
    sf.push(new StackFrame().addParam("n", n));

    while (!sf.isEmpty()) {
      StackFrame f = sf.pop();
      BigInteger pn = (BigInteger) f.get("n");
      if (pn.compareTo(BigInteger.ONE) <= 0) { continue; }
      else {
        sf.push(new StackFrame().addParam("n", pn.subtract(BigInteger.ONE)));
        res = res.multiply(pn);
      }
    }

    return res;
  }
}
```

With this, notice two things: first, we mimic the behavior of the call stack manually. This consequently means that, unless there is a method inside that uses the stack, we never push any activation records. Second, by managing the stack ourselves, we drastically increase the limit to the number of possible "recursive calls," since we push instances of our `StackFrame` onto the heap. Theoretically, we could continuously push new "frames" to our stack so long as we have active and available heap memory. This, of course, is impossible with current hardware limitations, so in due time, with a large-enough call to `factLoop`, the JVM terminates the program with an `OutOfMemory` exception. In the relevant test suite, we do not include tests for extraordinarily large numbers to preserve space, but we encourage the readers to try out such test cases, e.g., 100000000!. We should state that these tests will not complete in a reasonable amount of time.[1]

[1] On an AMD Ryzen 5 3600 with 16GB of DDR4 RAM, this test did not complete within a three hour time frame.

4.2 Object Mutation and Aliasing

A limitation that we have purposefully imposed on our object/class design is the inability to modify the values of instance variables. Value mutation is a foreign concept in some programming languages, but we have made extensive use of it throughout our time in the land of Java. In this section, we will discuss the implications of allowing instance variable mutation and how it can lead to some unexpected behavior.

To access a `private` instance variable, we design a `public` accessor method, which returns the instance variable. To modify a `private` instance variable, we design a `public` mutator method, which takes in a parameter and assigns the instance variable to the parameter. Let's return to the `Point` class to demonstrate. Suppose that we instantiate a point p to $(7, 4)$, but we then want to change or modify either coordinate. We can do so by calling the `setX` or `setY` methods, respectively. Testing setter methods is important to verify that a change occurred when invoking the setter/mutator method, which we confirm through the accessor method. Another way of phrasing this is that, when testing a mutator, we care about the *side-effect* of the method rather than what it returns, namely nothing. Setter methods, or methods that modify outside values or data are definitionally *impure*. Because we want to be able to alter an instance variable, these can no longer be marked as `final`, so we remove this keyword.[1]

Listing 4.35—Testing `Point` Setter Methods

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

class PointTester {

  @Test
  void testSetX() {
    Point p = new Point(7, 4);
    p.setX(3);
    assertEquals(3, p.getX());
  }

  @Test
  void testSetY() {
    Point p = new Point(7, 4);
    p.setY(2);
    assertEquals(2, p.getY());
  }
}
```

Listing 4.36—`Point` Class with Accessor and Mutator Methods

```
class Point {

  private int x;
  private int y;

  public Point(int x, int y) {
    this.x = x;
    this.y = y;
```

[1]This is not to suggest that we should never use `final` instance variables. In fact, we *should* use `final` instance variables whenever possible, since object mutation introduces the possibility of easy-to-overlook bugs.

```
  }

  public int getX() { return this.x; }

  public int getY() { return this.y; }

  public void setX(int x) { this.x = x; }

  public void setY(int y) { this.y = y; }
}
```

What are some consequences to mutating an object? One comes through the notion of *object aliasing*. Recall that objects point to references in memory. Therefore if we instantiate a `Point` p_1, then initialize another `Point` p_2 to reference p_1, then both objects refer to the same `Point` instance in memory. If we modify p_1 through a setter method, then p_2 will reflect the change.

Listing 4.37—Object Aliasing

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

class PointTester {

  @Test
  void pointAliasing() {
    Point p1 = new Point(7, 4);
    Point p2 = p1;
    p1.setX(3);
    assertEquals(3, p1.getX());
    assertEquals(3, p2.getX());
    assertEquals(p1, p2);
  }

  @Test
  void testSetY() {
    Point p1 = new Point(7, 4);
    Point p2 = p1;
    p1.setY(2);
    assertEquals(2, p1.getY());
    assertEquals(2, p2.getY());
    assertEquals(p1, p2);
  }
}
```

This idea carries over to other, more complex classes as well. For example, strings, arrays, lists, and others are all objects, and therefore, they are all subject to object aliasing. Modifying one `ArrayList` instance will modify all other `ArrayList` instances that reference the same object. This is a common source of bugs in Java programs, and it is important to be aware of this behavior. In the following example, we will demonstrate aliasing through the `ArrayList` data structure containing `Point` objects. We add a series of `Point` instances to an `ArrayList`, which is then aliased by another `ArrayList`. We then add another `Point` instance to the first `ArrayList`, followed by a verification that the lists are the same size. Additionally, we traverse over the lists and verify that the elements are the same through the == operator. Remember that == returns whether or not two objects reference the same instance in memory. Because these lists are merely aliases of each other, they will contain references to the same `Point` instances.

Listing 4.38—Testing `ArrayList` Aliasing

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

class PointTester {

  private final Point P1 = new Point(7, 4);
  private final Point P2 = new Point(3, 2);
  private final Point P3 = new Point(1, 8);

  @Test
  void pointArrayListAliasingTest() {
    ArrayList<Point> list1 = new ArrayList<>(List.of(P1, P2, P3));
    ArrayList<Point> list2 = list1;
    list1.add(new Point(5, 6));

    // First we can verify that the lists are actually the same.
    assertTrue(list1 == list2);

    // Size testing.
    assertTrue(list1.size() == list2.size());

    // Make sure both lists contain the same elements.
    for (int i = 0; i < list1.size(); i++) {
      assertTrue(list1.get(i) == list2.get(i));
    }
  }
}
```

Example 4.9. Now that we have classes, mutation, and accessibility, we can finally implement generic data structures such as an `ArrayList`. In this example we will implement the behavior of the `ArrayList` data structure, which means we finally understand what is going on under the hood in the Collections API. Let's design a class `MiniArrayList`, which operates over any type using generics. Like generic static methods, we must quantify the generic type, but unlike static methods, however, we quantify the type over the class declaration, meaning that all instance and static methods observe/respect the quantifier and do not need to be separately quantified.

In addition to the class header, what else does an `ArrayList` store? A backing array of elements and its corresponding length, of course! The array, as we described in Chapter 3, dynamically resizes as we add or insert elements. The logical size of the array, i.e., the number of presently-existing elements, is stored in `size`, whereas the current capacity, i.e., how many elements can currently be stored without a resize, is stored in `capacity`. Our class will provide two constructors: one that instantiates the backing array to store ten elements, and another that allows the user to specify. Interestingly, this shows off a great example of one constructor calling another of the same class, an idea called *constructor chaining*.

Listing 4.39—`MiniArrayList` Class

```
class MiniArrayList<T> {

  private T[] elements;
  private int size;
  private int capacity;
```

```
  public MiniArrayList() { this(10); }

  public MiniArrayList(int capacity) {
    this.size = 0;
    this.capacity = capacity;
  }
}
```

Notice that we declare an array of type `T` to store the elements of our mini array list. We now must instantiate the array inside the second constructor. The problem is that we cannot instantiate an array of a generic type. Recall that generics are a compile-time feature, and arrays are a runtime feature. Therefore, we cannot instantiate an array of a generic type. Instead, we must instantiate an array containing type `Object`, followed by a cast to contain type `T`.[1] This is called an *unchecked cast*, and it is a necessary evil in Java to support powerful classes that operate over generic arrays.

Listing 4.40—`MiniArrayList` Class Instantiating the Backing Array

```
class MiniArrayList<T> {

  private T[] elements;
  private int size;
  private int capacity;

  public MiniArrayList() { this(10); }

  public MiniArrayList(int capacity) {
    this.size = 0;
    this.capacity = capacity;
    this.elements = (T[]) new Object[capacity];
  }
}
```

We now need to implement the `add` method, which adds an element to the end of the array list. We first check if the array is full, and if so, we resize the array. We then add the element to the end of the array and increment the size. Resizing the array is, fortunately, not complicated; all we need to do is instantiate a new, larger array, copy the existing elements over, then reassign the instance variable. The question now is, by what factor should the array capacity increase? This decision is implementation-dependent, but we will use a doubling factor out of convenience. We make `resize` private because it is an implementation detail that the user does not need to know about. To write coherent tests, we should also write the `get` method, which returns the element at a given index, as well as `size`, which returns the number of logical elements in the list. For now, we will not worry about bounds checking, but we will return to this in a later chapter on exceptions.

Listing 4.41—Testing `MiniArrayList` add Method

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

class MiniArrayListTester {

  @Test
  void testAdd() {
    MiniArrayList<Integer> list = new MiniArrayList<>();
```

[1] To be pedantic, the array is of type `Object[]`, and we cast it to type `T[]`.

```
    list.add(100);
    list.add(200);
    list.add(300);
    assertAll(
      () -> assertEquals(3, list.size()),
      () -> assertEquals(100, list.get(0)),
      () -> assertEquals(200, list.get(1)),
      () -> assertEquals(300, list.get(2)));
  }
}
```

Listing 4.42—`MiniArrayList` Adding Elements

```
class MiniArrayList<T> {

  private static final int RESIZE_FACTOR = 2;

  /**
   * Adds an element to the end of the list.
   * @param element the element to add.
   */
  public void add(T element) {
    if (this.size == this.capacity) { this.resize(); }
    this.elements[this.size++] = element;
  }

  /**
   * Resizes the backing array by a factor specified by the class.
   */
  private void resize() {
    this.capacity *= RESIZE_FACTOR;
    T[] newArray = (T[]) new Object[this.capacity];
    for (int i = 0; i < this.size; i++) { newArray[i] = this.elements[i]; }
    this.elements = newArray;
  }

  public T get(int index) { return this.elements[index]; }

  public int size() { return this.size; }
}
```

We will write two more methods: `insert` and `remove`, which inserts an element e at a given index i, and removes an element e respectively. These two methods are similar in that they alter the backing array by shifting its values right and left. Accordingly, our implementation will contain the private helper methods `shiftRight` and `shiftLeft`. If we attempt to insert an element into a list that must be resized, we call `resize`. Both `insert` and `remove` warrant test cases! Like the `get` counterpart, neither of these new methods will perform bounds checking, so testing out-of-bounds behavior, for the time being, is not pertinent.

Listing 4.43—Testing `MiniArrayList` `insert` and `remove` Methods

```
import static Assertions.assertAll;

class MiniArrayListTester {

  @Test
  void testInsert() {
    MiniArrayList<Integer> list = new MiniArrayList<>();
```

```
    list.add(100);
    list.add(300);
    list.insert(1, 150);
    assertAll(
      () -> assertEquals(3, list.size()),
      () -> assertEquals(100, list.get(0)),
      () -> assertEquals(150, list.get(1)),
      () -> assertEquals(300, list.get(2)));
  }

  @Test
  void testRemove() {
    MiniArrayList<Integer> list = new MiniArrayList<>();
    list.add(100);
    list.add(300);
    list.remove(0);
    assertAll(
      () -> assertEquals(1, list.size()),
      () -> assertEquals(300, list.get(0)));
  }
}
```

Listing 4.44—`MiniArrayList` Inserting and Removing Elements

```
class MiniArrayList<T> {

  // ... other details not shown.

  /**
   * Inserts an element at the given index.
   * @param e the element to insert.
   * @param idx the index to insert at.
   */
  public void insert(T e, int idx) {
    if (this.size == capacity) { this.resize(); }
    this.shiftRight(idx);
    this.elements[idx] = e;
  }

  /**
   * Removes the element at the given index.
   * @param idx the index to remove.
   * @return the element removed.
   */
  public T remove(int idx) {
    T e = this.get(idx);
    this.shiftLeft(idx);
    this.size--;
    return e;
  }

  /**
   * Shifts all elements to the left of the given index one position leftwards.
   * Note that this method overwrites the element at the given index.
   * @param idx the index to shift left of.
   */
  private void shiftLeft(int idx) {
    for (int i = idx; i < this.size - 1; i++) {
      this.elements[i] = this.elements[i + 1];
    }
  }
```

```
  /**
   * Shifts all elements to the right of the given index one position rightwards.
   * @param idx the index to shift right of.
   */
  private void shiftRight(int idx) {
    for (int i = size - 1; i > idx; i--) {
      this.elements[i] = this.elements[i - 1];
    }
  }
}
```

Example 4.10. Let's see a few more examples of object aliasing and mutation. These examples will not be meaningful in what they represent, but are great exercises in testing your understanding. We will create five classes: `A`, `B`, `C`, `D`, and `E`. Class `A` contains one mutable string instance variable; its constructor assigns the instance variable to the parameter thereof. Classes `B` and `C` are identical aside from the name: they contain an immutable object of type `A` as an instance variable. Class `D` stores an integer array of ten elements. Finally, class `E` stores a mutable integer as an instance variable.

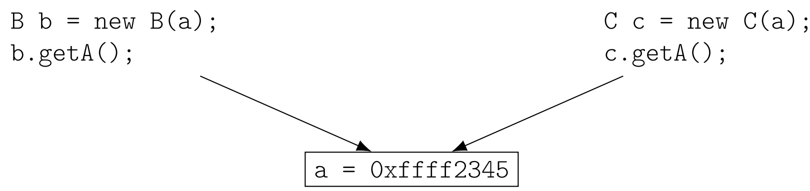
We present several test cases that assert different pieces of these classes. We will analyze each one and determine why it uses either `assertEquals` or `assertNotEquals` in its comparison. Our first series of tests only focuses on classes `A`, `B`, and `C` to keep things simple. We insert blanks in the assertion statements for you to fill in as exercise before checking your answers.

Listing 4.45—Testing Object Mutation and Aliasing

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

class ClassTester {

  @Test
  void testOne() {
    final A a = new A("Hello!");
    B b = new B(a);
    C c = new C(a);

    // What do each of these output?
    assert______(b.getA(), c.getA());
    assert______(b.getA().getS(), c.getA().getS());
    a.setS("Hi!");
    assert______(b.getA().getS(), c.getA().getS());
    b.getA().setS("howdy!");
    assert______(b.getA().getS(), c.getA().getS());

    B b2 = new B(a);
    assert______(a, b2.getA());
    assert______(b, b2);
    b = b2;
    assert______(a, b.getA());
    assert______(b, b2);
  }
}
```

To set the scene, we first declare `a` as an immutable instance of `A` with the string literal `"Hello!"`. Then, we instantiate objects `b` and `c` of types `B` and `C` respectively, each receiving `a` as an argument to their constructors.

Comparing `b.getA()` against `c.getA()` is a comparison of two references to the same object. Because `a` is immutable, we cannot change its value, so both `b` and `c` will always refer to the same object. Therefore, we use `assertEquals` to compare the two references. In particular, passing `a` to both constructors passes a reference to the same object.[1]



Comparing `b.getA().getS()` against `c.getA().getS()` is a comparison of two references to the same object, similar to the previous problem, right? Wrong! Recall that the `String` class overrides the `.equals` method implementation to compare strings for their content rather than their reference. Should we choose to compare the two strings for referential equality, we must use the == operator. In this case, we use `assertEquals` since the two strings are equal in content, but using == would also work because the strings are also equal in reference.

In the third line we change the value of the string inside `a` to be `"Hi!"`, which updates across all instances that point to `a`. Therefore, rerunning the same comparison as before still results in a true equality.

In the fifth line, we retrieve the `A` object instance pointed to by `B` and change its underlying string to be `"Howdy!"`. Rerunning the same test as before yet again results in a true equality. Because `b` points to the same `a` that `c` references, this change propagates across all references to `a`, even if we do not directly modify `a`.

We then declare a new instance of `B` named `b2`, which references the same `a` as before. If we check the value of `a` against the value of `a` inside `b2`, we of course get a true equality.

We immediately follow this comparison with one in which we compare `b` to `b2`. Because these are completely distinct object instantiations, the equality does not hold true.

Up next we reassign `b` to point to `b2`. This is a reassignment of a reference, not a reassignment of an object. Therefore if we check `b` against `b2` for equality, it is now trivially true.

Let's now test the `D` and `E` classes, which use arrays as instance variables.

Listing 4.46—Testing Object Mutation and Aliasing

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

class ClassTester {

  @Test
```

[1] In Figure ??, we use memory addresses to refer to the location of `a`. To be a bit pedantic, objects are not stored directly in system memory per se, but rather a location accessible by the Java Virtual Machine.

```
  void testTwo() {
    D d = new D();
    E e = new E(42);
    E[] arrOfE = new E[10];
    for (int i = 0; i < arrOfE.length; i++) { arrOfE[i] = new E(i); }
    assert______(arrOfE[2], arrOfE[5]);
    assert______(arrOfE[2].getNumber(), arrOfE[5].getNumber());

    for (int i = 0; i < arrOfE.length; i++) { arrOfE[i] = e; }
    assert______(arrOfE[0], arrOfE[2]);
    assert______(arrOfE[0].getNumber(), arrOfE[2].getNumber());
    arrOfE[7].setNumber(102);
    assert______(arrOfE[0].getNumber(), arrOfE[2].getNumber());
  }
}
```

The objects `d` and `e` are instantiated to types `D` and `E` respectively, the latter of which receives the integer `42` as an argument to the constructor. We follow this up with an array of ten `E` objects. The loop after instantiates each element of the array to a new `E` object with the integer `i` as an argument to the constructor.

So, what happens if we compare any arbitrary element e against any other arbitrary element e' such that $e \neq e'$? Because they are all instantiated to distinct instances of `E`, any equality comparison is false. We can extend this to retrieving the number inside each `E` object and comparing them. Because they are all distinct objects, and each `E` instance receives a different number, the equality is false.

The second for loop assigns each element of the array to the `e` object. Then, we can compare any arbitrary element against any other arbitrary element, and they will always be equal, since every element is a reference to the same memory reference.

Thus, if we set the number of one arbitrary element, this change propagates to every other element in the list, because again, all references point to the same object.

Example 4.11. Some may question why we emphasize aliasing and mutation. When working with the Collections API and designing data structures, proper care must be taken to avoid undesired behavior. Consider what happens if we design a class `F` that stores a reference to a `List<Integer>` as an instance variable. Then, suppose we instantiate two distinct instances of `F`, namely f_1 and f_2, each of which receive the same list of numbers. If we then mutate the list somewhere inside of f_1, then the list stored as a reference inside f_2 is also changed.

Listing 4.47—Testing List Mutation and Aliasing

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

class ClassTester {

  @Test
  void testListAliasing() {
    List<Integer> ls = new ArrayList<>(List.of(1, 2, 3, 4, 5));
    F f1 = new F(ls);
    F f2 = new F(ls);
    f1.getList().set(2, 100);
    assertEquals(100, f1.getList().get(2));
    assertEquals(100, f2.getList().get(2));
  }
```

```
}
```

Listing 4.48—Class Storing a List of Integers

```
import java.util.List;

class F {

  private final List<Integer> LS;

  public F(List<Integer> ls) { this.LS = ls; }

  public List<Integer> getList() { return this.LS; }
}
```

Example 4.12. Recall the `LinkedList` class from Chapter 3. If you have ever wondered how it works under the hood, now is the time to find out! We will design a *doubly-linked list* data structure that stores arbitrarily-typed elements.

First, remember the structure of a linked list: they are composed of nodes, which hold the data and a pointer to the next element in the chain/sequence. These types of linked lists are *singly-linked*, because nodes only refer to the successive element. In contrast, our class models a doubly-linked list, since its nodes point to their predecessor and their successor.

We need a generic class that stores references to the first and last elements of the list. Let's create the `DoublyLinkedList` class to receive a type parameter `T` and store the first and last nodes as instance variables. It's important to realize that, whoever uses this class will not be exposed to the innards of the class, i.e., how the links are established/constructed/altered/removed. We wish to preserve the idea of encapsulation, after all.

We run into an eminent problem when declaring the types of the instance variables: what should they be? We need to design a class that encapsulates the value of the node, and holds references to its previous and successor nodes. Some programmers may consider designing a separate `.java` file for this class, but remember the encapsulation methodology: nobody outside of this class should even be aware that nodes exist in the first place. So, we can create a private and static `Node<T>` class, which is local to the definition of `DoublyLinkedList`. A privatized class can only ever be static, because it does not make sense to say that a private class definition belongs to an arbitrary instance of the class in which it resides. We also override the `toString` to output the underlying stringified data of the node.

Listing 4.49—Doubly-Linked List Class

```
class DoublyLinkedList<T> {

  private static class Node<T> {

    private T value;
    private Node<T> prev;
    private Node<T> next;

    private Node(T value) {
      this.value = value;
    }

    @Override
    public String toString() {
```

```
      return this.value.toString();
    }
  }

  private Node<T> head;
  private Node<T> tail;

  public DoublyLinkedList() {
    this.last = this.first = null;
  }
}
```

Notice that, in the constructor of `DoublyLinkedList`, we assign the first and last references to each other, which both point to `null`. This is because, when the list is empty, there is no first or last element.

To test the methods that we are about to design, we will override the `toString` method (of `DoublyLinkedList`) to print the elements inside brackets, separated by commas and a space. To traverse over the list, however, we should use a custom-defined `Iterator`, which will be its own localized class definition. We have seen iterators before, but until now we have not implemented one on our own. The idea is, fortunately, very simple: we keep track of the current node, and upon calling `hasNext`, we return whether or not the node is `null`. Similarly, invoking `next` returns the value of the stored node and moves the pointer forward via the "next" instance. Finally, we create the `.iterator` method, which returns an instance of the iterator superclass. We do not want to expose how this particular iterator works, because the caller does not need to be aware of this logic; they are only concerned with iterating over the structure, in this case, a doubly-linked list.

Listing 4.50—Doubly-Linked List Iterator

```
import java.util.Iterator;

class DoublyLinkedList<T> {
  // ... previous code omitted ...

  public Iterator<T> iterator() {
    return new DoublyLinkedListIterator<>(this.first);
  }

  private static class DoublyLinkedListIterator<T> implements Iterator<T> {

    private Node<T> current;

    private DoublyLinkedListIterator(Node<T> first) {
      this.current = first;
    }

    @Override
    public boolean hasNext() {
      return this.current != null;
    }

    @Override
    public T next() {
      T value = this.current.value;
      this.current = this.current.next;
      return value;
    }
```

```
  }
}
```

Using the iterator in `toString` is straightforward: we have a while loop that continues until no more elements are present. We complete two tasks at the same time by having an iterator, which then makes subsequent traversals over the list easier.

Now we can write methods to add, retrieve, and remove elements from the list. To add an elemenet, we need to take the links of `first` and `last`, and reassign them accordingly to remain consistent with our doubly-linked list property. If the list is empty, then we just have to assign the new node *n* to both the `first` and `last` references. Otherwise, we set the "next" pointer of `last` to *n*, and set the "previous" pointer of *n* to `last`.

Listing 4.51—Testing the Doubly-Linked List Add Method

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

class DoublyLinkedListTester {

  @Test
  void testAdd() {
    DoublyLinkedList<Integer> list = new DoublyLinkedList<>();
    assertAll(
      () -> assertEquals("[]", list.toString()),
      () -> list.add(1),
      () -> list.add(2),
      () -> list.add(3),
      () -> assertEquals("[1, 2, 3]", list.toString()),
      () -> list.add(4),
      () -> list.add(1),
      () -> list.add(5),
      () -> assertEquals("[1, 2, 3, 4, 1, 5]", list.toString()));
  }
}
```

Listing 4.52—Doubly-Linked List Add Method

```java
class DoublyLinkedList<T> {
  // ...  previous code omitted ...

  /**
  * Adds a new node to the end of the list.
  * @param data The data to be stored in the new node.
  */
  public void add(T data) {
    Node<T> newNode = new Node<>(data);

    // If the list is empty, make the new node the first and last node.
    if (this.first == null) {
      this.first = newNode;
    } else {
      // Otherwise, add the new node to the end of the list.
      newNode.prev = this.last;
      this.last.next = newNode;
    }
    this.last = newNode;
  }
}
```

Retrieving an element is trivial, as it's just a matter of traversing over the list and returning the data at the index of a node. If the index is out of bounds, we return an empty `Optional`.[1]

Listing 4.53—Testing the Doubly-Linked List Get Method

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

class DoublyLinkedListTester {

  @Test
  void testGet() {
    DoublyLinkedList<Integer> list = new DoublyLinkedList<>();
    assertAll(
      () -> assertEquals(Optional.empty(), list.get(0)),
      () -> list.add(50),
      () -> list.add(25),
      () -> list.add(100),
      () -> assertEquals(Optional.of(50), list.get(0)),
      () -> assertEquals(Optional.of(25), list.get(1)),
      () -> assertEquals(Optional.of(100), list.get(2)),
      () -> assertEquals(Optional.empty(), list.get(3)),
      () -> list.add(1000),
      () -> list.add(10000),
      () -> list.add(50),
      () -> assertEquals(Optional.of(1000), list.get(3)),
      () -> assertEquals(Optional.of(10000), list.get(4)),
      () -> assertEquals(Optional.of(50), list.get(5)),
      () -> assertEquals(Optional.empty(), list.get(6)));
  }
}
```

Listing 4.54—Doubly-Linked List Get Method

```java
import java.util.Optional;

class DoublyLinkedList<T> {
  // ... previous code omitted ...

  /**
   * Returns the element at a given index as an Optional.
   * @param idx index to retrieve.
   * @return Optional.empty() if the index is out of bounds,
   * the data at that node's index otherwise.
   */
  public Optional<T> get(int idx) {
    Node<T> curr = this.first;
    int i = 0;
    while (curr != null && i < idx) {
      curr = curr.next;
      i++;
    }
    return idx >= 0 && curr != null
            ? Optional.of(curr.data)
            : Optional.empty();
  }
}
```

[1] It is, in general, a better idea to use exceptions, but we have not covered them yet.

Finally we arrive at element removal, which is not as cut-and-dry. We want to pass the element-to-remove (compared via `equals`), but we need to adjust the pointers accordingly. In particular, there are four cases to consider:

(a) If the element-to-remove e is the first of the list, then its successor is now the first. Its previous pointer is adjusted to now point to `null`.

(b) If the element-to-remove e is the last of the list, then its predecessor is now the last. Its next pointer is adjusted to now point to `null`.

(c) If the element to remove e is neither the first nor the last, we retrieve its previous node p, its next node n, and assign $p_{next} = n$, and $n_{prev} = p$. This, in effect, "delinks" e from the list, which gets consumed by the garbage collector.

(d) If the element-to-remove e is not in the list, do nothing.

Listing 4.55—Testing the Doubly-Linked List Remove Method

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

class DoublyLinkedListTester {

  @Test
  void testRemove() {
    DoublyLinkedList<Integer> list = new DoublyLinkedList<>();
    assertAll(
      () -> list.add(50),
      () -> list.add(25),
      () -> list.add(100),
      () -> list.remove(50),
      () -> assertEquals("[25, 100]", list.toString()),
      () -> list.remove(100),
      () -> assertEquals("[25]", list.toString()),
      () -> list.remove(25),
      () -> assertEquals("[]", list.toString()),
      () -> list.remove(25),
      () -> assertEquals("[]", list.toString()));
  }
}
```

Listing 4.56—Doubly-Linked List Remove Method

```
class DoublyLinkedList<T> {
  // ...  previous code omitted ...

  /**
   * Removes an element from the linked list, if it exists.
   * @param data value to be removed, compared via .equals.
   */
  public void remove(T data) {
    Node<T> curr = this.first;
    while (curr != null) {
      if (curr.data.equals(data)) {
        // Case 1:  if it's the first.
        if (curr == this.first) {
          curr.next = this.first.next;
          this.first = curr.next;
        }
        // Case 2:  if it's the last.
```

```
        else if (curr == this.last) {
          curr.prev.next = null;
          this.last = curr.prev;
        }
        // Case 3:  if it's anything else.
        else {
          curr.prev.next = curr.next;
          curr.next.prev = curr.prev;
        }
        break;
      } else {
        curr = curr.next;
      }
    }
  }
}
```

Example 4.13. Some programming languages do not come standard with data structures such as `Map` or, if they do, they are cumbersome to utilize. A substitute for the common mapping data structure is called an *association list*, originating with the Lisp programming language. Its desired purpose is nearly identical to that of a map, but with worse performance implications. In this example we will design such a structure, as if `Map` did not exist in Java.

Associations lists, as their name implies, associate values to other values, just like a map. In dynamically-typed languages, e.g., Scheme, association lists accept any type as their key and any type as their value. Therefore, we could have an association list that maps a string to an integer, or an integer to a list of strings, and so on. Should we want to use truly arbitrary types in the list, we can assign `Object` to both key and value types.

Our association list will support several methods that are related to their functional programming eqiuvalents. In particular, we want a `lookup` method to retrieve the associated value of some element and an an `extend` method to add a new association. Note that the `extend` method will, rather than modifying the current association list, return a new association list with the new association added. This is because we want to preserve the idea of immutability, which is a common theme in functoinal programming. Association lists, therefore, need to have a "parent" pointer to keep track of those associations in the list that we extend from.[1] We will also override the `toString` method to print the associations in a readable format.

Listing 4.57—Association List Tester

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

class AssociationListTester {

  @Test
  void testAssociationList() {
    AssociationList<String, Integer> list = new AssociationList<>();
    assertAll(
      () -> assertEquals("[]", list.toString()),
      () -> list = list.extend("a", 1),
      () -> assertEquals("[(a, 1)]", list.toString()),
      () -> list = list.extend("b", 2),
```

[1] In the next section on abstract classes and interpreters, we will revisit this idea in greater detail.

```
      () -> assertEquals("[(b, 2), (a, 1)]", list.toString()),
      () -> list = list.extend("c", 3),
      () -> assertEquals("[(c, 3), (b, 2), (a, 1)]", list.toString()),
      () -> assertEquals(Optional.of(3), list.lookup("c")),
      () -> assertEquals(Optional.of(2), list.lookup("b")),
      () -> assertEquals(Optional.of(1), list.lookup("a")),
      () -> assertEquals(Optional.empty(), list.lookup("d")));
  }
}
```

Listing 4.58—Association List Class

```
import java.lang.StringBuilder;
import java.util.Optional;

class AssociationList<K, V> {

  private final K key;
  private final V value;
  private final AssociationList<K, V> parent;

  public AssociationList() {
    this.key = null;
    this.value = null;
    this.parent = null;
  }

  private AssociationList(K key, V value, AssociationList<K, V> parent) {
    this.key = key;
    this.value = value;
    this.parent = parent;
  }

  /**
   * Returns the value associated with a given key.
   * @param key the key to lookup.
   * @return the value associated with the key, if it exists.
   */
  public Optional<V> lookup(K key) {
    AssociationList<K, V> curr = this;
    while (curr != null) {
      if (curr.key.equals(key)) {
        return Optional.of(curr.value);
      } else {
        curr = curr.parent;
      }
    }
    return Optional.empty();
  }

  /**
   * Adds a new association to the list.
   * @param key the key to associate.
   * @param value the value to associate.
   * @return a new association list with the new association.
   */
   public AssociationList<K, V> extend(K key, V value) {
    return new AssociationList<>(key, value, this);
   }

  @Override
```

```
  public String toString() {
    StringBuilder sb = new StringBuilder("[");
    AssociationList<K, V> curr = this;
    while (curr != null) {
      sb.append(String.format("(%s, %s)", curr.key, curr.value));
      curr = curr.parent;
      if (curr != null) { sb.append(", "); }
    }
    sb.append("]");
    return sb.toString();
  }
}
```

Each association list in our representation stores exactly one association. Each time we extend the association, we create a new list that points to the previous list. This is a very inefficient way to store associations, but it is a common functional way of extending bindings in a programming language, most often in simple interpreted languages.

Chapter Exercises

Exercise 4.1. (⋆)
Design the `Car` class, which stores a `String` representing the car's make, a `String` representing the car's model, and an `int` representing the car's year. Its constructor should receive these three values and store them in the instance variables. Be sure to write instance accessor and mutator methods for modifying all three fields. That is, you should write `getMake()`, `setMake(String s)`, and so forth, to access and modify the fields directly.

Exercise 4.2. (⋆)
Design the `Dog` class, which stores a `String` representing the breed, a `String` representing its name, and an `int` representing its age in years. You should also store a boolean to keep track of whether or not the dog is a puppy. A dog is a puppy if it is less than two years old. Its constructor should receive these three values and store them in the instance variables. Be sure to write instance accessor and mutator methods for modifying all three fields. That is, you should write `getBreed()`, `setBreed(String s)`, and so forth, to access and modify the fields directly.

Exercise 4.3. (⋆)
Design the `Person` class, which stores a `String` representing the person's first name, a `String` representing the person's last name, and an `int` representing the person's age in years. Its constructor should receive these three values and store them in the instance variables. Be sure to write instance accessor and mutator methods for modifying all three fields. That is, you should write `getFirstName()`, `setFirstName(String s)`, and so forth, to access and modify the fields directly.

Exercise 4.4. (⋆)
Design the `Employee` class, which stores the employee's first and last names as strings, their birthyear as an integer, their yearly salary as a double (we will assume that all employees are paid some value greater than zero), and their employee ID as a string. To make things interesting, assume that an employee's ID is not alterable and must be set in the constructor. The employee ID is constructed using the first five characters of their last name, the first letter of their first name, and the last two digits of their birthyear. For instance, if the employee's name is `Joshua Crotts` and their birthyear is `1999`, their employee ID is `CrottJ99`. Its constructor should receive the name, birthyear, and salary as parameters, but build the employee ID from the name and birthyear. Be sure to design the relevant accessor and mutator methods.

Exercise 4.5. (⋆)
As part of the `Employee` class, design the `void bonus()` method, which updates the salary of an employee. Calling `bonus` on an employee increases their salary by ten percent.

Exercise 4.6. (⋆⋆)
As part of the `Employee` class, override the `equals` and `toString` methods from the `Object` class to compare two employees by their employee ID and to print the employee's name, birthyear, salary, and employee ID respectively separated by commas.

Exercise 4.7. (⋆⋆)
In this exercise you will design a class for storing employees. This relies on completing the `Employee` class exercise.

(a) Design the `Job` class, which stores a list of employees `ArrayList<Employee>` as an instance variable. Its constructor should receive no arguments.

(b) Design the `void addEmployee(Employee e)` method, which adds an employee to the `Job`.

(c) Design the `void removeEmployee(Employee e)` method, which removes an employee from the `Job`.

(d) Design the `Optional<Double> computeAverageSalary()` method, which returns the average salary of all employees in the `Job`. If there are no employees, return an empty `Optional`.

(e) Design the `Optional<Employee> highestPaid()` method, which returns the employee whose salary is the highest of all employees in the `Job`. If there are no employees, return an empty `Optional`.

(f) Override the `toString` method to print out the list of employees in the `Job`. You can use the default `toString` implementation of the `ArrayList` class.

Exercise 4.8. ($\star\star$)
In this exercise you will design a *linear congruential generator*: a pseudorandom number generation algorithm. In particular, the C programming language standard library defines two functions: `rand` and `srand`. The latter sets the *seed* for the generator, and `rand` returns a random integer between $[0, 2^{15})$. The formula for this generator is a recurrence relation:

$$
\begin{aligned}
next &= |r_n \cdot 1103515245 + 12345| \\
r_{n+1} &= \left(\frac{next}{2^{16}}\right) \ \% \ 2^{15};
\end{aligned}
$$

(a) Design the `LcgRandom` class, which implements this behavior. In particular, it should have two constructors: one that receives a seed value s, and another that sets the seed to one. The seed initializes the value of r_0.

(b) Design the `int genInt()` method, which returns a random integer between 0 and 2^{15} using this algorithm.

(c) Design the `IntStream stream()` method, which returns a stream of random numbers that uses `genInt` to generate numbers. Hint: use `generate`!

(d) Design the `genInt(int b)` method that returns an integer between $[0, b]$. Note that $0 \leq b < 2^{15}$; you do not need to account for values outside of this range. Do **not** simply loop until you find a value between that range; instead, use modulus to your advantage.

Exercise 4.9. ($\star\star\star$)
This question has six parts.

(a) Design the `Matrix` class, which stores a two-dimensional array of integers. Its constructor should receive two integers m and n representing the number of rows and columns respectively, as well as a two-dimensional array of integers. Copy the integers from the passed array into an instance variable array.

(b) Design the `void set(int i, int j, int val)` method, which sets the value at row i and column j to `val`.

(c) Design the `void add(Matrix m)` method, which adds the values of the passed matrix to the current matrix. If the dimensions of the passed matrix do not match the dimensions of the current matrix, do nothing.

(d) Design the `void multiply(Matrix m)` method, which multiplies the values of the passed matrix to the current matrix. If we cannot multiply m with this matrix, do nothing.

(e) Design the `void transpose()` method, which transposes the matrix. That is, the rows become the columns and the columns become the rows. You may need to alter the dimensions of the matrix.

(f) Design the `void rotate()` method, rotates the matrix 90 degrees clockwise. To rotate a matrix, compute the transposition and then reverse the rows. You may need to alter the dimensions of the matrix.

(g) Override the `String toString()` method to print out the matrix in a boxed format.

Exercise 4.10. (★★)
This exercise has five parts.

(a) Design the `GameObject` class, which stores a `Pair<Double, Double>` denoting its center (x, y) position and a `Pair<Double, Double>` denoting its width and height respectively. Its constructor should receive four `double` values representing x, y, *width*, and *height*. Be sure to write instance accessor and mutator methods for modifying both fields. That is, you should write `double getLocationX()`, `void setLocationX(double d)`, and so forth, to access and modify the `Pair` values directly.

(b) Design the `boolean collidesWith(GameObject obj)` method that returns if `this GameObject` collides with the parameter `obj`. You should design this solution as if the game objects are shaped like rectangles (which they are!).

(c) Design the `double distance(GameObject obj)` method that returns the Euclidean distance from the center of `this GameObject` to the center of the parameter `obj`.

(d) Design the `double move(double dx, double dy)` method that moves the object about the Cartesian (two-dimensional) plane. The distance should be a delta represented as two `double` numbers `dx` and `dy` that directly manipulate the object position. For instance, if `dx` is 3.0 and `dy` is -2.0 and the object is currently at $< 2.0, -9.0 >$, invoking `move(3.0, -2.0)` updates the object to be at $< 5.0, -11.0 >$.

(e) Override the `String toString()` method to call the `toString` methods of the two instance variables, conjoined by a semicolon.

Exercise 4.11. (★★)
This exercise has three parts.

(a) Design the `GameRunner` class, which stores a list of objects `ArrayList<GameObject>` as an instance variable. Its constructor should receive an integer representing a random number generator seed. It should first instantiate `rand` to a new `Random` object with this seed, and then populate the list with twenty random `GameObject` instances at random **integer** positions with random **integer** sizes. These random positions should be between $[-10, 10]$ for both coordinates and the random sizes should be between $[1, 10]$ for both dimensions.

(b) Design the `void moveObjects()` method, which moves each object by three positive x units and four negative y units.

(c) Design the `String stringifyObjects()` method, which converts each object in the list into its string representation, with brackets around the elements, and separated by commas. Hint: you can use one method from the `Stream` class to do this quickly!

Exercise 4.12. (⋆⋆)
This exercise involves the "Twenty-One" game implementation from the chapter.

(a) Change each card to use the Unicode symbol counterpart rather than the `"X of Y"` `toString` model, where `X` is the value and `Y` is the suit. The Unicode symbols are available on the second page of this PDF: `https://www.unicode.org/charts/PDF/U1F0A0.pdf`. This will be a little tedious, but it makes the game look cooler!

(b) Add the Ace, Jack, Queen, and King cards, instead of the previous implementation of using four cards whose values were all ten. A simple solution is to use a `String` that keeps track of the "name" of a card alongside the other instance variables.

(c) Add an AI to the game (you do not need to test this class). This involves writing the `AI` class and designing the `boolean play(Deck deck)` method. An `AI` has a `ArrayList<Card>`, similar to `Player`, but makes decisions autonomously using the following algorithm (written in a pseudocode-like language):

```
boolean play(Deck d) {
  score = getScore()
  if score < 16 then:
    cards.add(d.drawCard())
    return true;
  else if (score > 16 && score < 21) {
    k = Generate a random integer between [0, 3).
    if k is zero then:
      cards.add(d.drawCard())
      return true;
  }
  return false;
}
```

The method returns whether or not the AI drew a card. If they did not draw a card, then their turn is over. When playing the game, the player can see the first two cards dealt to an AI, but nothing more. You might want to add a static variable to the `Card` class representing the "covered card." Note that the AI knows only the context of its deck of cards; it is not aware of any other `Player` or `AI`.

(d) After designing the `AI` class and adding one to your game, create an `ArrayList<AI>` simulating multiple computer players in the game.

Exercise 4.13. (⋆)
Add the `void set(T e, int idx)` method to the `MiniArrayList` class, which sets the element at *idx* to the given *e* element.

Exercise 4.14. (⋆)
Add the `void isEmpty()` method to the `MiniArrayList` class, which returns whether or not the list is empty.

Exercise 4.15. (⋆⋆)
Add the `void clear()` method to the `MiniArrayList` class, which "removes" all elements from the list. This should not change the capacity of the list. Note that there's a reason why "removes" is in quotes. We rank this exercise as a two-star not because of its length, but because it is a little tricky.

Exercise 4.16. (⋆⋆)
Override the `equals` method (from `Object`) in the `MiniArrayList` class to compare two lists by their elements. Return `true` if all elements in the two lists are `.equals` to one another, and `false` otherwise.

Exercise 4.17. (⋆⋆)
Using the `StackFrame` class, design an implementation of the tail recursive factorial method. Recall how to do this from Chapter 2: instead of pushing an activation record to the call stack, we can simply update the bindings in the existing frame.

Exercise 4.18. (⋆⋆⋆)
This exercise has seven parts.

A *chunked array list* data structure avoids the overhead of copying the underlying array upon running out of free spots. The idea is to break the collection into chunks, namely, as an `ArrayList` of arrays. Assuming that the underlying collection of chunks is adequately populated, this collection will seldom require a resizing operation. This data structure will not support arbitrary insertions or removals.

(a) Design the generic `ChunkedArrayList` class. It should store, as an instance variable, an `ArrayList<T[]>` of chunks, where T is the parameterized type. Design two constructors: one that receives a chunk size s and a number of preallocated chunks n, and another constructor that receives no parameters, defaulting n to 10 and s to 50.

(b) Design the `void add(T t)` method that, when given an item t, adds it to the end of the current chunk. If we run out of space in the current chunk, add it to the next chunk in succession. If there are no available chunks, add a new `T[]` of size s to the list. Hint: use modulus.

(c) Design the `T get(int i)` method that, when given an index i, returns the item at that index. The user of this data structure should not need to know about the chunks or their implementation. Therefore, if $s = 10$, and we access index 27, it should receive the element in chunk 3, index 7. Assume that i is in bounds.

(d) Design the `void resizeChunks(int n)` method that resizes each chunk to the input argument n. Depending on this value, you will need to either reallocate each underlying array or shift values around. For example, if we have a chunk array list with 150 elements whose chunks hold up to 50 elements each, and we resize the chunks to be 25 in maximum capacity, we will double the number of necessary chunks. On the other hand, if we resize the chunks to hold 100 elements, then the values in chunk two are shifted into chunk one, and those in chunk three are shifted into chunk two.

(e) Design the `int getChunkCapacity()` method that returns the maximum capacity of each chunk.

(f) Design the `int size()` method that returns the total number of elements in the chunk array list.

(g) Design the `int getChunkSize()` method that returns the number of chunks currently in-use.

Exercise 4.19. (⋆⋆⋆)
This exercise has seven parts.

A *persistent data structure* is one that saves intermittent data structures after applying operations that would otherwise alter the contents of the data structure. Take, for instance, a standard FIFO queue. When we invoke its 'enqueue' method, we modify the underlying data structure to now contain the new element. If this were a persistent queue, then enqueueing a new element would, instead, return a new queue that contains all elements and the newly-enqueued value, thereby leaving the original queue unchanged.

(a) First, design the generic, private, and static class `Node` inside a generic `PQueue` class skeleton. It should store, as instance variables, a pointer to its next element as well as its associated value.

(b) Then, design the `PQueue` class, which represents a persistent queue data structure. As instance variables, store "first" and "last" pointers as `Node` objects, as well as an integer to represent the number of existing elements. In the constructor, instantiate the pointers to `null` and the number of elements to zero.

(c) Design the `PQueue<T> enqueue(T t)` method that enqueues a value onto the end of a new queue containing all the old elements, in addition to the new value.

(d) Design the `T dequeue()` method that removes the first element of the queue, returning a new queue without this first value.

(e) Design the `T peek()` method that returns the first element of the queue.

(f) Design the `PQueue<T> of(T... vals)` method that creates a queue with the values passed as `vals`. Note that this must be a variadic method. Do not create a series of `PQueue` objects by enqueueing each element into a distinct queue; this is incredibly inefficient. Instead, allocate each `Node` one-by-one, thereby never calling `enqueue`.

(g) Design the `int size()` method that returns the number of elements in the queue. You should not traverse the queue to compute this value.

5. Advanced Object-Oriented Programming

5.1 Interfaces

Interfaces are a way of grouping classes together by a ubiquitous behavior. We have worked with interfaces before without acknowledging their properties as an interface. For example, the `Comparable` interface is implemented by classes that can be compared against each other. In particular, there is a single method that must be implemented by any class that implements the `Comparable` interface: the `compareTo` method. The `compareTo` method takes in a single argument of the same type as the class that implements the `Comparable` interface and returns an integer. Said integer is negative if the object instance is less than the argument, zero if the object instance is equal to the argument, and positive if the object instance is greater than the argument.

So, by having a class implement the `Comparable` interface, we group it into that subset of classes that are, indeed, comparable. Doing so implies that these classes have an ordering and are sortable in, for example, a Java collection.

In addition to the `Comparable` interface, we have worked with the `List`, `Queue`, and `Map` interfaces, which all have a set of methods that must be implemented by any class that implements the interface. Recall that `ArrayList` and `LinkedList` are both types of `List` objects, and this interface describes several methods that all lists, by definition, must override. To *override* a method means that we provide a new implementation of the method that is different from the default implementation provided by the interface.

Defining an Interface

Example 5.1. Imagine that we want to design an interface that describes a shape. All (two-dimensional) shapes have an area and a perimeter, so we can define an interface that, when implemented by a class, requires that the class provide an implementation of the `area` and `perimeter` methods. A common convention for user-defined interfaces is to prefix the names with `I` to distinguish them from classes. Moreover, the names of interfaces are either nouns or, more traditionally, verbs, since they describe behaviors or characteristics of a class. [1]

Listing 5.1—Shape Interface

```
interface IShape {

  double area();

  double perimeter();
}
```

We cannot write any tests for the `IShape` interface directly, since it is impossible to instantiate an interface. We can, however, write two different classes that implement `IShape`, and test those. To demonstrate, we will write and test the `Pentagon` and `Octagon` classes whose constructors receive (and then store as instance variables) the side length of the shape. Fortunately, the definitions thereof are trivial because they are nothing more than regurgitations of the mathematical formulae. Notice that, when testing, we initialize the object instance to be of type `IShape`, not `Pentagon` or `Octagon`. This is because we want to be able to categorize these classes as types of `IShape` instances rather than solely instances of `Pentagon` or `Octagon` respectively. This is a common practice in object-oriented programming, and it is called *polymorphism*. Polymorphism is the ability of an object to take on many forms. In this case, the `IShape` interface is the form that the `Pentagon` and `Octagon` classes use to take on the form of a shape as we described.

Listing 5.2—Shape Tester

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

class IShapeTester {

  private static final DELTA = 0.01;

  @Test
  void testPentagon() {
    IShape p1 = new Pentagon(1);
    IShape p2 = new Pentagon(7.25);
    assertAll(
      () -> assertEquals(1.72, p1.area(), DELTA),
      () -> assertEquals(90.43, p2.area(), DELTA),
      () -> assertEquals(5, p1.perimeter(), DELTA),
      () -> assertEquals(36.25, p2.perimeter(), DELTA));
  }

  @Test
  void testOctagon() {
    IShape o1 = new Octagon(1);
    IShape o2 = new Octagon(7.25);
    assertAll(
      () -> assertEquals(4.83, o1.area(), DELTA),
      () -> assertEquals(253.79, o2.area(), DELTA),
      () -> assertEquals(8, o1.perimeter(), DELTA),
      () -> assertEquals(58, o2.perimeter(), DELTA));
```

[1] We do not add the `public` keyword to the interface definition because all interface methods are implicitly public.

```
  }
}
```

Listing 5.3—Pentagon Class

```
class Pentagon implements IShape {

  private final double SIDE_LENGTH;

  public Pentagon(double sideLength) { this.SIDE_LENGTH = sideLength; }

  @Override
  public double area() {
    return 0.25 * Math.sqrt(5 * (5 + 2 * Math.sqrt(5)))
              * Math.pow(this.SIDE_LENGTH, 2);
  }

  @Override
  public double perimeter() {
    return 5 * this.SIDE_LENGTH;
  }
}
```

Listing 5.4—Octagon Class

```
class Octagon implements IShape {

  private final double SIDE_LENGTH;

  public Octagon(double sideLength) { this.SIDE_LENGTH = sideLength; }

  @Override
  public double area() {
    return 2 * (1 + Math.sqrt(2)) * Math.pow(this.SIDE_LENGTH, 2);
  }

  @Override
  public double perimeter() {
    return 8 * this.SIDE_LENGTH;
  }
}
```

Example 5.2. Recall from the previous chapter our "Twenty-one" card game example. In that, we designed the `Suit` class, which contained four public and static instances of `Suit`, where each represented one of the four valid card suits. This design works as intended, it fails to be elegant and demonstrate how the suits are all the same, but differ only in their string representation. Let's now design the `ISuit` interface, which requires any implementing class to override the `stringify` method.

Listing 5.5—Suit Interface Example

```
interface ISuit {

  /**
   * Returns the string representation of the suit.
   */
  String stringify();
}
```

From here, we can define four separate classes, all of which implement `ISuit` and override the `stringify` method. These classes are incredibly simple, and as such, we will show only the `Diamond` and `Heart` classes.

Listing 5.6—Diamond Suit Class

```java
class Diamond implements ISuit {

  public Diamond() {}

  @Override
  public String stringify() { return "♢"; }
}
```

Listing 5.7—Heart Suit Class

```java
class Heart implements ISuit {

  public Heart() {}

  @Override
  public String stringify() { return "♡"; }
}
```

As we see, both `Diamond` and `Heart` implement `ISuit` and handle "stringification" differently. We can test these definitions by storing a list of `ISuit` instances and ensuring that the correct character is returned.

Listing 5.8—Suit Interface Tester

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

import java.util.List;
import java.util.ArrayList;

class ISuitTester {

  @Test
  void suitTester() {
    List<ISuit> suit = new ArrayList<>();
    // Add diamonds at even indices, hearts at odd indices.
    for (int i = 0; i < 100; i++) {
      if (i % 2 == 0) { suit.add(new Diamond()); }
      else { suit.add(new Heart()); }
    }

    // Now check to verify that the stringification works.
    for (int i = 0; i < suit.size(); i++) {
      if (i % 2 == 0) { assertEquals("♢", suit.get(i).stringify()); }
      else { assertEquals("♡", suit.get(i).stringify()); }
    }
  }
}
```

One extra piece of information that we should share is that we can instantiate objects in different ways. To demonstrate why this matters, suppose we initialize an object s_1 to type `ISuit`, but instantiate it as type `Diamond`. Then, we initialize another object s_2 to type Diamond and instantiate it as type Diamond. We would expect that s_1 and s_2 are equivalent, but this is not the case. Suppose Diamond contains a method diamondCount that does something irrelevant, but belongs solely to the Diamond class. Because s_1 is of type ISuit, we cannot invoke the diamondCount method, since ISuit knows nothing about said method. On the contrary, s_2 can certainly invoke diamondCount, but it is not polymorphic, since it is not of type ISuit. Should we want to be able to invoke diamondCount on s_1, we need to *downcast* s_1 to type Diamond.

```
ISuit s1 = new Diamond();
s1.diamondCount();              // Compile-time error!
Diamond s2 = new Diamond();
s2.diamondCount();              // Works but not polymorphic.
((Diamond) s1).diamondCount(); // Works but downcasts.
```

Example 5.3. Animals are a common example of an interface. Imagine that, in our domain of animals, every animal can speak one way or another. Speaking involves returning a string representing the sound that the animal makes. By designing the IAnimal interface, we can group all animals that can speak together. We can then design classes that implement the IAnimal interface and provide an implementation of the speak method. When testing the latter, we can write tests that instantiate a collection of IAnimal instances, and invoke speak on each of them polymorphically. In doing so we get a refresher of the Java stream API.

Listing 5.9—Animal Interface

```
interface IAnimal {

  /**
   * Returns the sound that the animal makes.
   */
  String speak();
}
```

Listing 5.10—Animal Tester

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

import java.util.List;
import java.util.ArrayList;

class IAnimalTester {

  @Test
  void testCat() {
    IAnimal cat = new Cat();
    assertEquals("Meow!", cat.speak());
  }

  @Test
  void testDog() {
    IAnimal dog = new Dog();
    assertEquals("Woof!", dog.speak());
  }

  @Test
  void testListOfAnimals() {
```

```
    List<IAnimal> animals = new ArrayList<>();
    animals.add(new Cat());
    animals.add(new Dog());
    animals.add(new Cat());
    assertEquals("[Meow!, Wolf! Meow!]",
                 animals.stream()
                        .map(IAnimal::speak)
                        .collect(Collectors.toList()));
  }
}
```

Listing 5.11—Cat Class

```
class Cat implements IAnimal {

  @Override
  public String speak() { return "Meow!"; }
}
```

Listing 5.12—Dog Class

```
class Dog implements IAnimal {

  @Override
  public String speak() { return "Woof!"; }
}
```

Example 5.4. Suppose we want to design an interface that boxes an arbitrary value. We have seen this idea through autoboxing and autounboxing of the primitive datatypes and the wrapper classes, but our interface extends the concept to any type. We can define an interface that requires that any class that implements it provide an implementation of the `box`, `get`, and `set` methods. Boxing a value means that we can pass it around as a reference rather than as a raw value. Recall that passing primitives to methods is by value and, therefore, the method cannot change the value of the primitive. If, however, we box the primitive, then we can pass the boxed value to a method and change the value of the boxed value. We will first design the generic `IBox` interface, and then we will design a class that implements the methods.

Interestingly, interfaces may have static methods. Our `IBox` interface has a static `box` method that returns a box of the type passed in as an argument. This is useful because we can call the `box` method without having to instantiate a class that implements the `IBox` interface. We can then use the `get` and `set` methods to retrieve and change the value of the box.

Listing 5.13—Box Interface

```
class IBox<T> {

  /**
   * Boxes the value of type T.
   */
  static IBox<T> box(T t);

  /**
   * Returns the boxed value of type T.
   */
  T get();
```

```
  /**
   * Sets the boxed value of type T.
   */
  void set(T t);
}
```

Listing 5.14—Box Tester

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

class IBoxTester {

  private static <T> void modifyBox(IBox<T> box, T t) {
    box.set(t);
  }

  @Test
  void testIntegerBox() {
    IBox<Integer> box = IntegerBox.box(5);
    assertAll(
      () -> assertEquals(5, box.get()),
      () -> modifyBox(box, 10),
      () -> assertEquals(10, box.get()));
  }
}
```

Listing 5.15—Integer Box Class

```
class IntegerBox implements IBox<Integer> {

  private Integer value;

  private IntegerBox(Integer value) {
    this.value = value;
  }

  @Override
  public static IBox<Integer> box(Integer value) {
    return new IntegerBox(value);
  }

  @Override
  public Integer get() { return this.value; }

  @Override
  public void set(Integer value) { this.value = value; }
}
```

The Java Swing API is a graphics framework for designing graphical interfaces and drawing shapes/images. In addition to these capabilities, it also supports user input through the keyboard, mouse, and other means. Compared to a class like `Scanner`, which waits for the user to press “Enter” when they are finished typing input, the Swing API allows for dynamic input and is constantly monitored by the program. We call the part of the program that listens and processes events an *event listener*. A popular example is the `ActionListener` interface, which is used to listen for a broad classification of events. The `ActionListener` interface has a single method, `actionPerformed`, that is invoked when an event occurs. The `actionPerformed` method receives an `ActionEvent` object that contains information about the event that occurred, which is then usable by the method to determine what to do in response to the event. Because graphical interface design goes beyond the scope of this textbook, we will omit a code example, but we mention action listeners to demonstrate that interfaces are not limited to the examples we have seen thus far. Moreover, the Swing API provides more specific listeners for processing keyboard and mouse events, e.g., `KeyListener`, `MouseListener`, `MouseMotionListener`, and so forth. We could, for instance, design a class that implements the `MouseListener` interface and provides an overriding implementation of the `mouseClicked` method. Then, inside a Java Swing graphical component, we might hook the class as a mouse listener and, when the user clicks the mouse, the `mouseClicked` method is invoked.

Example 5.5. An amazing insight into the power of interfaces is already present in Java, but deriving it ourselves is useful. Consider the notion of first-class functions: the concept in which functions and data are equivalent, and we can pass functions around as arguments and return them from other functions. In Java, we can pass functions around as arguments, mimicing first-class functions, by designing a *functional interface*.

Let's design the generic `Function<T, V>` interface, which quantifies over two types `T`, representing the input type, and `V`, representing the output type. The `Function<T, V>` interface has a single static method, `apply`, that receives an argument of type `T` and returns a value of type `V`. We can then design a class that implements the `Function<T, V>` interface and provides an implementation of the `apply` method. We can then pass the class around as an argument to other methods, and invoke the `apply` method on the class to get the result of the function. An incredibly simple example is `AddOne`, which implements the `Function<Integer, Integer>` interface and adds one to its input. We make the constructor of the implementing class private to prevent any unnecessary instantiations; we only want to use the class as a first-class function rather than an object.

Listing 5.16—Functional Interface

```java
interface Function<T, V> {

  static V apply(T t);
}
```

Listing 5.17—Add One Functional Interface Tester

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

class AddOneTester {

  @Test
  void addOneTester() {
    assertAll(
      () -> assertEquals(0, AddOne.apply(1)),
      () -> assertEquals(3, AddOne.apply(2)),
      () -> assertEquals(30001, AddOne.apply(30000)));
  }
}
```

Listing 5.18—Add One Functional Interface Implementation

```
class AddOne implements Function<Integer, Integer> {

  private AddOne() {}

  @Override
  public static Integer apply(Integer i) { return i + 1; }
}
```

So far, we have not demonstrated the potential of first-class functions in Java with our design. Suppose we have a list of `Integer` values $l = v_1, v_2, \ldots, v_n$ and a function f, and we want to apply f to each element thereof. That is, we will create a new list $l' = f(v_1), f(v_2), \ldots, f(v_n)$. Normally, we would need to write a specific function for each function f, but by passing a functional interface, we can write a single method that receives this list and a function f and applies f to each element of the list. This operation, in general, is called `map`, which we saw during our discussion on streams![1]

Listing 5.19—Map Function Tester

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

import java.util.List;

class FunctionMapTester {

  @Test
  void testMap() {
    List<Integer> l = List.of(1, 2, 3, 4, 5);
    Function<Integer, Integer> addOne = new AddOne();
    assertAll(
      () -> assertEquals(List.of(2, 3, 4, 5, 6), FunctionMap.map(l, addOne)),
      () -> assertEquals(List.of(), FunctionMap.map(List.of(), addOne)));
  }
}
```

Listing 5.20—Map Function Implementation

```
import java.util.List;
import java.util.ArrayList;

class FunctionMap {

  /**
   * Applies the function f to each element of the list l.
   * @param l the list of elements.
   * @param f the function to apply to each element.
   * @return the list of elements after applying f to each element.
   */
  public static <T, V> List<V> map(List<T> l, Function<T, V> f) {
    return l.stream()
            .map(t -> f.apply(t))
            .collect(Collectors.toList());
  }
}
```

[1] Do not confuse this with the concept of a map/dictionary from our data structures/collections discussion.

Example 5.6. Java 8 introduced the `Function` interface, so we do not have to design our own version. Using it, we do not need to design a separate `AddOne` class to implement the interface; we can make use of method referencing via the `::` operator. Let's rewrite the `addOne` example doing so. We will also show off the fact that Java will auto-box and unbox an integer primitive into its `Integer` counterpart, meaning that our `addOne` method does not need to receive and return objects, but rather primitives instead, which are easier to work with. Moreover, lambda expressions are passable to methods that receive `Function` arguments, because Java automatically converts them into `Function` objects, mimicing the autoboxing treatment of primitive datatypes.[1]

Listing 5.21—Map Function Tester

```java
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static FunctionMap.map;

import java.util.List;

class FunctionMapTester {

  static int addOne(int i) { return i + 1; }

  @Test
  void testMap() {
    List<Integer> l = List.of(1, 2, 3, 4, 5);
    assertAll(
      () -> assertEquals(List.of(2, 3, 4, 5, 6), map(l, FunctionMapTester::addOne)
          ),
      () -> assertEquals(List.of(), map(List.of(), FunctionMapTester::addOne)),
      () -> assertEquals(List.of(2, 3, 4, 5, 6), map(l, i -> i + 1)),
      () -> assertEquals(List.of(2, 3, 4, 5, 6), map(List.of(), i -> i + 1)));
  }
}
```

Example 5.7. Now that we have interfaces, we can write a very simple expression tree interpreter! What do we mean by this? Consider the arithmetic expression $5 + (3 + 4)$. According to the standard order-of-operations, we evaluate the parenthesized sub-expressions first, then reduce outer expressions. So, in our case, we add 3 and 4 to get 7, followed by an addition of 5. We can represent this idea as an evaluation tree, where we travel from bottom-up, evaluating sub-expressions as they are encountered. How does this relate to interfaces? Suppose we create the `IExpr` interface, which contains a single method: `int value`, which returns the value of an expression.

Listing 5.22—Expression Interface

```java
interface IExpr {

  /**
   * Returns the value of the expression.
   */
  int value();
}
```

[1]In the tester code snippet below, we could omit the `FunctionMapTester::` type qualification because the method is defined inside the same class that it is used.

The simplest (atomic) values in our language are numbers, or literals as they are called. A `Lit` stores a single integer as an instance variable, and returns this instance variable upon a `value` invocation, which means `Lit` must implement the `IExpr` interface. Testing this class is trivial, so we will only write two tests.

Listing 5.23—Literal Tester

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

class LitTester {

  @Test
  void testLit() {
    assertAll(
      () -> assertEquals(0, new Lit(0).value()),
      () -> assertEquals(42, new Lit(42).value()));
  }
}
```

Listing 5.24—Lit Class Implementation

```
class Lit implements IExpr {

  private final int N;

  public Lit(int n) { this.N = n; }

  public int value() { return this.N; }
}
```

How do we add two numbers? Or, rather, how do we represent the addition of two numbers? This question comes through the answer to our question of representing literal values. Addition expressions store two `IExpr` expressions as instance variables, and (mutually) recursively calls their `value` methods, followed by a summation. Note the parallelism to how we do this when evaluating expressions either on paper or in our heads.

Listing 5.25—Addition Expression Tester

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

class AddTester {

  @Test
  void testAdd() {
    assertAll(
      () -> assertEquals(12, new Add(new Lit(5), new Add(new Lit(3), new Lit(4))))
          ,
      () -> assertEquals(42, new Add(new Lit(41), new Lit(1))),
      () -> assertEquals(101, new Add(new Add(new Lit(123), new Lit(-43)),
                                     new Add(new Lit(2), new Lit(19)))));
  }
}
```

Listing 5.26—Addition Expression Implementation

```
class Add implements IExpr {

  private final IExpr LHS;
  private final IExpr RHS;

  public Add(IExpr lhs, IExpr rhs) {
    this.LHS = lhs;
    this.RHS = rhs;
  }

  @Override
  public int value() { return this.LHS.value() + this.RHS.value(); }
}
```

Thus we have a programming language that interprets numbers and addition expressions! We could add more elements/operators to this language, and we encourage the readers to get creative.

Example 5.8. Symbolic differentiators are programs that take a mathematical expression and compute its derivative, but non-numerically. That is, it examines and interprets the structure of the expression to calculate the derivative. In this example we will write a symbolic differentiator in Java using interfaces and classes. Note that you do not need any calculus knowledge to follow along.

The formal definition of the derivative of a function is not a necessary detail to concern ourselves of; but in short, it measures the instantaneous rate-of-change at a given point of the function, i.e., the slope of the line tangent to the point. There are several rules for computing derivatives of functions, all of which are common exercises in an introductory calculus course. We want to be able to construct expressions in such a way that it is trivial to differentiate their components. As an example, the derivative of the expression $3x^2 - 16x + 100$ is $6x - 6$ due to specific rules that we will explain shortly. The idea, however, is that we have a large expression to find the derivative of, and by differentiating its subcomponents, we obtain the derivative of the larger. Let's see what all we need to do.

First, let's design the `Expression` interface, which contains one method to compute the derivative of an `Expression`: `Expression derivative(String var)`. Expressions in calculus are differentiated with respect to a given variable, e.g., x, so we need to pass that to any expression that we wish to differentiate. Now, any class that implements `Expression` must override `derivative`.

Using some basic calculus derivative shortcuts/rules, we can easily think of two more types of expressions: constants (e.g., 3, 0, 27) and monomials (e.g., ax^n where a, n are integers). So, let's design the `ConstantExpression` and `MonomialExpression` classes, the former of which has a constructor that receives a single integer c, whereas the latter stores the variable, the coefficient a, and finally the exponent n. To make working with these expressions easier, as well as ensuring testibility, we will override the `equals`, `hashCode`, and `toString` methods.

The derivative of a constant c is always zero, because the slope of a straight line, namely $f(x) = c$ is zero, i.e., non-changing. On the other hand, a monomial follows a different rule based off its coefficient and exponent: the derivative of ax^n is anx^{n-1} for any $n > 1$. If $n = 1$, then this trivially becomes a constant. There is one more edge-case to consider:

if the given variable v does not match the variable of the monomial, then the derivative is zero since it is differentiating with respect to a free variable.

Listing 5.27—Testing Constant and Monomial Derivatives

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

class DerivativeTester {

  @Test
  void testNumberExpressionDerivative() {
    assertAll(
      () -> assertEquals(new NumberExpression(0),
                         new NumberExpression(0).derivative("x")),
      () -> assertEquals(new NumberExpression(0),
                         new NumberExpression(10).derivative("x")));
  }

  @Test
  void testMonomialExpressionDerivative() {
    assertAll(
      () -> assertEquals(new ConstantExpression(3),
                         new MonomialExpression("x", 3, 1).derivative("x")),
      () -> assertEquals(new ConstantExpression(0),
                         new MonomialExpression("x", 3, 10).derivative("y")),
      () -> assertEquals(new MonomialExpression("x", 6, 1),
                         new MonomialExpression("x", 3, 2).derivative("x")));
  }
}
```

Listing 5.28

```java
import java.util.Objects;

class ConstantExpression implements Expression {

  private final int CONSTANT;

  public ConstantExpression(int c) { this.CONSTANT = c; }

  @Override
  public Expression derivative(String v) { return new ConstantExpression(0); }

  @Override
  public boolean equals(Object obj) {
    if (!(obj instanceof ConstantExpression)) { return false; }
    else { return ((ConstantExpression) obj).CONSTANT == this.CONSTANT; }
  }

  @Override
  public int hashCode() { return Objects.hash(this.CONSTANT); }

  @Override
  public String toString() { return String.format("%d", this.CONSTANT); }
}
```

Listing 5.29

```
import java.util.Objects;

class MonomialExpression implements Expression {

  private final int COEFFICIENT;
  private final int EXPT;
  private final String VAR;

  public MonomialExpression(String var, int a, int n) {
    this.VAR = var;
    this.COEFFICIENT = a;
    this.EXPT = n;
  }

  @Override
  public Expression derivative(String v) {
    if (this.VAR.equals(v)) {
      if (this.EXPT == 1) {
        return new ConstantExpression(this.COEFFICIENT);
      } else {
        return new MonomialExpression(this.VAR, this.COEFFICIENT * this.EXPT,
                                      this.EXPT - 1);
      }
    } else {
      return new ConstantExpression(0);
    }
  }

  @Override
  public boolean equals(Object obj) {
    if (!(obj instanceof MonomialExpression)) {
      return false;
    } else {
      MonomialExpression expr = (MonomialExpression) obj;
      return this.VAR.equals(expr.VAR)
          && this.COEFFICIENT == expr.COEFFICIENT
          && this.EXPT == expr.EXPT;
    }
  }

  @Override
  public int hashCode() {
    return Objects.hash(this.VAR, this.COEFFICIENT, this.EXPT);
  }

  @Override
  public String toString() {
    return String.format("%d%s^%d", this.COEFFICIENT, this.VAR, this.EXPT);
  }
}
```

Let's move into compositional expressions: expressions that contain expressions as instance variables. Such an example is an additive operator; the derivative of the expression $(f(x) + g(x))' = f'(x) + g'(x)$, where f' is the derivative of f. In summary, the derivative of a sum is the sum of the derivatives of its operands. To represent sequential operands, e.g., $x + y + z + \cdots + w$, we will store the expressions in a list. Note that our symbolic differentiator neither simplifies expressions nor combines like terms.

Listing 5.30—Testing Additive Derivatives

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

class DerivativeTester {

  // ...  other testing methods not shown.

  @Test
  void testAddExpressionDerivative() {
    assertAll(
      () -> assertEquals(
              new AddExpression(
                new MonomialExpression("x", 3, 2),
                new MonomialExpression("x", 6, 5)),
              new AddExpression(
                new MonomialExpression("x", 1, 3),
                new MonomialExpression("x", 1, 6)).derivative("x")),
      () -> assertEquals(
              new AddExpression(
                new MonomialExpression("x", 10, 4,
                new MonomialExpression("x", 12, 2),
                new MonomialExpression("x", -14, 1),
                new NumberExpression(6),
                new NumberExpression(0))),
              new AddExpression(
                new MonomialExpression("x", 2, 5),
                new MonomialExpression("x", 4, 3),
                new MonomialExpression("x", -7, 2),
                new MonomialExpression("x", 6, 1),
                new NumberExpression(9)).derivative("x")));
  }
}
```

Listing 5.31

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.Objects;
import java.util.stream.Collectors;

class AddExpression implements Expression {

  private final List<Expression> EXPR_LIST;

  public AddExpression(Expression... exprs) {
    this.EXPR_LIST = Arrays.asList(exprs);
  }

  public AddExpression(List<Expression> exprs) { this.EXPR_LIST = exprs; }

  @Override
  public Expression derivative(String var) {
    List<Expression> exprs = new ArrayList<>();
    this.EXPR_LIST.forEach(e -> exprs.add(e.derivative(var)));
    return new AddExpression(exprs);
  }
```

```
  @Override
  public boolean equals(Object obj) {
    if (!(obj instanceof AddExpression)) {
      return false;
    } else {
      AddExpression expr = (AddExpression) obj;
      for (int i = 0; i < this.EXPR_LIST.size(); i++) {
        if (!this.EXPR_LIST.get(i).equals(expr.EXPR_LIST.get(i))) {
          return false;
        }
      }
      return true;
    }
  }

  @Override
  public int hashCode() { this.EXPR_LIST.hashCode(); }

  @Override
  public String toString() {
    return this.EXPR_LIST.stream()
                       .map(Object::toString)
                       .collect(Collectors.joining(" + "));
  }
}
```

5.2 Inheritance

Classes may relate to other classes by a hierarchy. In particular, one class, called the *subclass*, can extend another class, called the *superclass*. A subclass inherits all the public methods and fields from its superclass. Classes can only extend one class at a time, unlike other programming languages such as C++.

Example 5.9. Suppose we have the `Alien` class defined as follows, which can move forward by one unit and turn left by 90 degrees in some world that it resides within.[1]

Listing 5.32—Alien Tester

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

class AlienTester {

  @Test
  void testAlien() {
    Alien r1 = new Alien();
    assertAll(
      () -> r1.moveForward(),
      () -> assertEquals(1, r1.getX()),
      () -> r1.turnLeft(),
      () -> assertEquals(Alien.Direction.NORTH, r1.getDir()),
      () -> r1.moveForward(),
      () -> assertEquals(1, r1.getY()),
      () -> r1.turnLeft(),
      () -> assertEquals(Alien.Direction.WEST, r1.getDir()),
      () -> r1.moveForward(),
      () -> assertEquals(0, r1.getX()),
```

[1] We base this example off of Karel J. Robot from [Bergin et al., 2013] and [Pattis, 1995].

```
        () -> r1.turnLeft(),
        () -> assertEquals(Alien.Direction.SOUTH, r1.getDir()),
        () -> r1.moveForward(),
        () -> assertEquals(0, r1.getY()),
        () -> r1.turnLeft(),
        () -> assertEquals(Alien.Direction.EAST, r1.getDir()),
        () -> r1.moveForward(),
        () -> assertEquals(1, r1.getX()));
  }
}
```

Listing 5.33—Alien Class Definition

```
class Alien {

  public enum Direction { NORTH, SOUTH, EAST, WEST };

  private int x;
  private int y;

  private Direction dir;

  public Alien() {
    this.x = 0;
    this.y = 0;
    this.dir = Direction.EAST;
  }

  /**
   * Moves the alien forward by one unit in the direction it is facing.
   */
  public void moveForward() {
    switch (this.dir) {
      NORTH -> this.y++;
      SOUTH -> this.y--;
      EAST -> this.x++;
      WEST -> this.x--;
    }
  }

  /**
   * Turns the alien left by 90 degrees.
   */
  public void turnLeft() {
    switch (this.dir) {
      NORTH -> this.dir = Direction.WEST;
      SOUTH -> this.dir = Direction.EAST;
      EAST -> this.dir = Direction.NORTH;
      WEST -> this.dir = Direction.SOUTH;
    }
  }

  // Accessors and mutators omitted for brevity.
}
```

What we have defined is an incredibly primitive alien class that stores its position and direction in a two-dimensional plane. Testing the alien, as we have done, is straightforward, but even such a simple alien definition must turn left three times to mimic the behavior of turning right once. We should extend the `Alien` class to add a `turnRight` method. We will call this class `RightAlien`, which adds a single method: `turnRight`. The other methods remain the same, since we do not want to overwrite their behavior. One important thing to note is that we invoke the superclass constructor via the `super()` invocation. We do so because we want the direction, x and y variables to be correctly initialized when instantiating an instance of `RightAlien`. As we will demonstrate with future examples, invoking the superclass constructor can be done with arguments.

Listing 5.34—Right Alien Tester

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

class RightAlienTester {

  @Test
  void testMoverAlien() {
    RightAlien r1 = new RightAlien();
    assertAll(
      () -> r1.turnRight(),
      () -> assertEquals(RightAlien.Direction.SOUTH, r1.getDir()),
      () -> r1.turnRight(),
      () -> assertEquals(RightAlien.Direction.WEST, r1.getDir()),
      () -> r1.turnLeft(),
      () -> assertEquals(RightAlien.Direction.SOUTH, r1.getDir()));
  }
}
```

Listing 5.35—Right Alien Class Definition

```
class RightAlien extends Alien {

  public RightAlien() { super(); }

  /**
   * Turns the Alien right by 90 degrees.
   */
  public void turnRight() {
    switch (this.getDir()) {
      NORTH -> this.setDir(Direction.EAST);
      SOUTH -> this.setDir(Direction.WEST);
      EAST -> this.setDir(Direction.SOUTH);
      WEST -> this.setDir(Direction.NORTH);
    }
  }
}
```

Great, we can turn right with this flavor of the alien! Though, moving forward by one unit is absurdly slow, so let's now design the `MileMoverAlien` class, which moves ten units for every `moveForward` call. A mile, in this two-dimensional world, is equal to ten units. Because we want to override the functionality of `moveForward` from `Alien`, we must redefine the method in the subclass, and add the `@Override` annotation. Moreover, we define this particular version of `moveForward` in terms of `moveForward` from the superclass. This is a common pattern when overriding methods: we want to reuse the functionality of the superclass, but add some additional behavior. In this case, we want to move ten units forward, instead of one. In order to invoke the superclass definition of `moveForward`, we prefix the method call with '`super.`', rather than '`this.`'. Should we accidentally prefix the method call with '`this.`', we would be invoking the subclass definition of `moveForward`, resulting in an infinite loop![1] One could make the case and say that this is, in fact, a form of recursion, and indeed this is true, but it is nonsensical recursion because the outcome not only undesired but also never terminates.

Listing 5.36—Mile Mover Alien Tester

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

class MileMoverAlienTester {

  @Test
  void testMileMoverAlien() {
    Alien r1 = new MileMoverAlien();
    assertAll(
      () -> r1.moveForward(),
      () -> assertEquals(10, r1.getX()),
      () -> r1.turnLeft(),
      () -> r1.moveForward(),
      () -> assertEquals(10, r1.getY()));
  }
}
```

Listing 5.37—Mile Mover Alien Class Definition

```java
class MileMoverAlien extends Alien {

  public MileMoverAlien() { super(); }

  @Override
  public void moveForward() {
    for (int i = 0; i < 10; i++) { super.moveForward(); }
  }
}
```

Now suppose we want a alien that "bounces" throughout the world. A bouncing alien will pick a random direction to face, then move two spots in that direction, simulating a bounce. Because the alien chooses a random direction, testing its implementation is difficult without predetermined knowledge of the random number generator. Therefore we will omit a tester for this class. All we must do is override the `moveForward` method, and invoke `super.moveForward` twice after facing a random direction.

Listing 5.38—Bouncer alien Class Definition

```java
import java.util.Random;

class BouncerAlien extends Alien {

  private final Random RNG;

  public BouncerAlien() {
    super();
    this.RNG = new Random();
  }
```

[1] Omitting '`this.`' still causes the method to infinitely loop, since not having the qualifier will cause Java to look in the current class definition.

```
  @Override
  public void moveForward() {
    switch (this.rand.nextInt(4)) {
      case 0: { this.setDir(Direction.NORTH); break; }
      case 1: { this.setDir(Direction.SOUTH); break; }
      case 2: { this.setDir(Direction.EAST); break; }
      case 3: { this.setDir(Direction.WEST); break; }
    }
    super.moveForward();
    super.moveForward();
  }
}
```

Why not create a world for this alien to live within, and objects to interact with or collide into? Let's design the `World` class, which stores a two-dimensional array of `WorldPosition` instances. The `WorldPosition` class is a very general wrapper class to store what we will call `WorldObject` instances. Because a `WorldObject` is not very specific, we will expand upon its implementation with a single subclass, that being `StarObject`. Stars are objects that an alien in the world can pick up and drop.

This is a lot of information to consider, so let's back up a bit and start by designing the `WorldPosition` class. A `WorldPosition` contains a list of `WorldObject` instances. Therefore, we know that `WorldPosition` encapsulates objects that exist on that particular position. We also need to write the `WorldObject` class, which does nothing but acts as a placeholder for other objects to extend; one of those being `StarObject`.

We, ideally, want aliens to be able to pick and place stars on a world position. It is nonsensical, though, for a aliens to pick stars on a `WorldPosition` that has no existing stars. Therefore, in the `WorldPosition` class, we will write a method that returns the number of instances of a given object. Doing so raises a question of, "How do we specify a class to count?," the answer to which comes via *reflection*.

Reflection is a programming language feature that allows us to inspect the structure of a class at runtime. We can use reflection to determine the class of an object, and then compare that class to the class we are using to search through the data structure. If the classes instances match (i.e., an object in the list is an instance of the desired searching class), in the case of our "counter" method, we increment the counter. To access an object's class information through reflection, we use the `getClass` method, which returns a `Class` instance. To receive any type of class as the parameter to a method, we parameterize the type of `Class` with a wildcard, <?>.

Why are we worrying about reflection in the first place? Would it not be easier to simply write a method that, say, returns the number of `StarObject` instances in the `WorldPosition` through perhaps an enumeration describing the type of object? The answer is a resounding yes, but forcing the programmer to write an enumeration just to describe the type of some class is cumbersome and unnecessary. Moreover, when we want to extend the functionality to include a new type, we must update the enumeration, which is a poor design choice. Reflection allows us to write a single method that can count the number of instances of any class, without having to continuously/repeatedly rewrite the method.

Listing 5.39—World Object Class Definition

```
class WorldObject {
```

```
  public WorldObject() {}
}
```

Listing 5.40—Star Object Class Definition

```
class StarObject extends WorldObject {

  public StarObject() { super(); }
}
```

Listing 5.41—World Position Class Definition

```
import java.util.ArrayList;
import java.util.List;
import java.lang.Class;

class WorldPosition {

  private List<WorldObject> WORLD;

  public WorldPosition() {
    super();
    this.WORLD = new ArrayList<>();
  }

  /**
   * Using streams, returns the number of occurrences of a given class type.
   * @param cls the class to search for.
   * @return those instances of a class that exist on the position.
   */
  public int count(Class<?> cls) {
    return this.WORLD.stream()
                     .filter(o -> obj.getClass().equals(cls))
                     .count();
  }
}
```

Finally we arrive at the `World` class. Perhaps we make it a design choice to disallow extension of this class. To block a class from being extended, we label it as `final` . The `World` class stores, as we stated, a two-dimensional array of `WorldPosition` instances, simulating a two-dimensional grid structure (where the plane origin lies in the top-left rather than the traditional bottom-left). Our constructor receives two integers denoting the width and height of the world, corresponding to the number of rows and columns of the underlying array, respectively.[1] Each position in the world is directly instantiated thereof to prevent later null pointer references. Said `World` class contains two methods: `addObject`, and `countStars`, where the former adds an object to a given position in the world, and the latter counts the number of stars on that position.

Listing 5.42—World Class Implementation

```
final class World {

  private final WorldPosition[][] WORLD;
```

[1]Because `WorldPosition` *is* a list, we can conclude that `World` *is* a two-dimensional array, meaning a programmer might ask about extending the functionality of an array. Arrays are not extendable, because they are not classes/interfaces.

```
  public World(int width, int height) {
    this.WORLD = new WorldPosition[width][height];
    for (int i = 0; i < width; i++) {
      for (int j = 0; j < height; j++) {
        this.WORLD[i][j] = new WorldPosition();
      }
    }
  }

  /**
   * Assigns a WorldObject to a given position in the world by adding
   * it to the list of objects.
   * @param obj the object to assign.
   * @param x the x-coordinate of the position.
   * @param y the y-coordinate of the position.
   */
  public void add(WorldObject obj, int x, int y) {
    this.WORLD[x][y].add(obj);
  }

  /**
   * Counts the number of stars on a given position in the world.
   * @return the number of stars.
   */
  public int countStars() {
    return this.WORLD[x][y].count(StarObject.class);
  }
}
```

5.3 Abstract Classes

We consider a class to be abstract if it is not representable by any instance. That is, we cannot create an instance of an abstract class. Abstract classes are useful when we want to define a class that is a generalization of other classes, but we do not want to create instances of the generalization.

Example 5.10. Consider, once again, a hierarchy of animals. There is no such thing as an "animal" or something that is solely called an animal. On the other hand, everything that we would categorize as an animal *is* an animal. Therefore it makes sense to say that animals are a generalization of other types of "sub"-animals. Imagine we want to write an `Animal` class, where we will say that any animal can speak. The abstract class contains a superfluous constructor as well as an abstract `speak` method. We define `speak` as abstract to denote that an animal can speak, but it is nonsensical for `Animal` to speak. Because it is impossible to instantiate an instance of `Animal`, it is similarly impossible to reasonably define `speak`.

Listing 5.43—Animal Class

```
abstract class Animal {

  public Animal() {}

  public abstract String speak();
}
```

Let's declare two subclasses: `Dog` and `Cat`, representing dogs and cats respectively. A cat can meow via the string `"Meow!"`, whereas a dog woofs via the string `"Woof!"`.

Listing 5.44—Dog Class

```
class Dog extends Animal {

  public Dog() { super(); }

  @Override
  public String speak() { return "Woof!"; }
}
```

Listing 5.45—Cat Class

```
class Cat extends Animal {

  public Cat() { super(); }

  @Override
  public String speak() { return "Meow!"; }
}
```

It might seem strange to use an abstract class, since we could write a `Speakable` interface to do the same logic. The differences between abstract classes and interfaces is a blurry line to beginning Java programmers (and even to some who have been programming for years), but in essence, we use abstract classes when we want to enforce a class hierarchy of "is-a" relationships, e.g., a `Cat` is-a `Animal`, and a `Dog` is-a `Animal`. Moreover, abstract classes can contain non-abstract methods, meaning that a subclass needs not to override such methods. Interfaces, on the other hand, contain only methods that the implementing class must override. In addition to the method distinction, abstract classes may contain instance variables, whereas interfaces may not.[1]

Example 5.11. Suppose we're writing a two-dimensional game that has different types of interactable objects in the world. The core game object stores the (x, y) location, with nothing more. Again, we want to design a class that specific types of game objects can extend. For instance, our game might contain circular and rectangular objects. Of course, circles and rectangles have different dimension units, namely radius versus width and height respectively. We plan for each object to be interactable with one another. Unfortunately, collision detection is a complicated set of algorithms whose discussion far exceeds the scope of this text. Conversely, there is an extremely straightforward solution that involves treating all objects as rectangles. We call this technique *axis-aligned bounding box.* Because not every object may be collidable, we will design a class `AxisAlignedBoundingBoxObject` that separately stores the object width and height as the dimensions of the bounding box. This class defines a method for colliding with another `AxisAlignedBoundingBoxObject`, which determines whether some point of o_1 is inside the bounding box of the o_2 object. This logic is not the focal point of the discussion, so we will only illustrate the example via an image and not explain the code itself. The purpose for this example is to demonstrate object hierarchy; not recreate the next best-selling two-dimensional side-scroller.

[1] Both abstract classes and interfaces can contain static methods and variables.

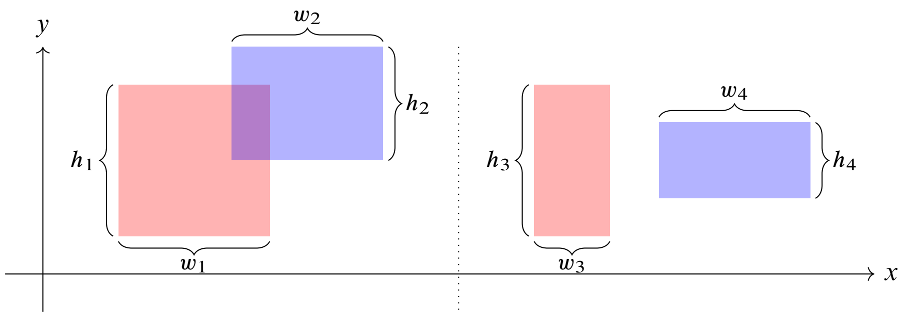


Figure 5.1: Collision Detection Between Rectangles.

Listing 5.46—Game Object Class Definition

```
abstract class GameObject {

  private int x;
  private int y;

  public GameObject(int x, int y) {
    this.x = x;
    this.y = y;
  }

  // Getters and setters omitted.
}
```

Listing 5.47—Axis-Aligned Bounding Box Class

```
abstract class AxisAlignedBoundingBoxObject extends GameObject {

  private int width;
  private int height;

  public AxisAlignedBoundingBoxObject(int x, int y, int width, int height) {
    super(x, y);
    this.width = width;
    this.height = height;
  }

  /**
   * Determines whether this object collides with another
       AxisAlignedBoundingBoxObject.
   * @param obj instance of AxisAlignedBoundingBoxObject.
   * @return true if the objects overlap and false otherwise.
   */
  public boolean collidesWith(AxisAlignedBoundingBoxObject obj) {
    return (this.getX() < obj.getX() + obj.width) &&
           (this.getX() + this.width >= obj.getX()) &&
           (this.getY() < obj.getY() + obj.height) &&
           (this.getY() + this.height >= obj.height);
  }

  // Getters and setters omitted.
}
```

We declared an abstract class to extend another abstract class; which is perfectly acceptable. Because it makes no sense to have an entity called `AxisAlignedBoundingBoxObject`, we declare it as abstract, but we need it to contain the functionality of `GameObject`, which calls for the inheritance. Normally, we would immediately write an extensive test suite for `collidesWith`, but because we cannot instantiate an `AxisAlignedBoundingBox` directly, we cannot test `collidesWith` at the moment. In a couple of paragraphs, however, this will be possible, with the additions of `CircleObject` and `RectangleObject`.

Listing 5.48—Testing AABB Collision Detection

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

class AxisAlignedBoundingBoxObjectTester {

  @Test
  void testCollidesWith() {
    AxisAlignedBoundingBoxObject o1 = new RectangleObject(30, 30, 1000, 2000);
    AxisAlignedBoundingBoxObject o2 = new RectangleObject(0, 0, 5, 5);
    AxisAlignedBoundingBoxObject o3 = new RectangleObject(400, 200, 750, 250);
    AxisAlignedBoundingBoxObject o4 = new RectangleObject(300, 100, 300, 200);
    AxisAlignedBoundingBoxObject o5 = new CircleObject(20, 30, 1000);
    AxisAlignedBoundingBoxObject o6 = new CircleObject(200, 250, 500);
    AxisAlignedBoundingBoxObject o7 = new CircleObject(30, 300, 1500);
    AxisAlignedBoundingBoxObject o8 = new CircleObject(90, 85, 200);
    assertAll(
      () -> assertTrue(o1.collidesWith(o2)),
      () -> assertTrue(o1.collidesWith(o4)),
      () -> assertTrue(o2.collidesWith(o3)),
      () -> assertTrue(o2.collidesWith(o8)),
      () -> assertTrue(o3.collidesWith(o5)),
      () -> assertTrue(o5.collidesWith(o4)),
      () -> assertTrue(o6.collidesWith(o7)),
      () -> assertTrue(o7.collidesWith(o3)));
  }
}
```

We need to translate our circles into axis-aligned bounding box, but what does that mean? In short, we convert the given radius into the corresponding diameter, and designate this diameter as the width and height of the bounding box. Rectangular objects, on other hand, need no such fancy translation, since a bounding box is a rectangle. Neither subclasses storetheir dimensions as instance variables, due to the fact that the superclass takes care of this for us.

The question that we anticipate many readers are thinking of is, why do we even distinguish objects of differing "types" if they both implement collision detection in the same fashion? Since we are working in the context of a game, the way we draw these objects will almost certainly be different! Let's, for the sake of emphasizing the distinctions, design a `IDrawable` interface, which provides one method: `draw(Graphics2D g2d)`, which gives us a `Graphics2D` object. We will not discuss, nor do we really care about the innards of a graphics library aside from the fact that it contains two primitive methods: `drawOval(int x, int y, int w, int h)` and `drawRect(int x, int y, int w, int h)`. Therefore our two object subclasses will implement `IDrawable` and override the method differently.

Listing 5.49—Drawable Interface Definition

```
interface IDrawable {

  /**
   * Provides a means of drawing primitive graphics.
   * The inner details of "Graphics2D" are not important to us;
   * we care about the fact that we can use the following methods:
   *
   * - drawOval(int x, int y, int w, int h);
   * - drawRect(int x, int y, int w, int h);
   */
  void draw(Graphics2D g2d);
}
```

Listing 5.50—Circle Object Class Definition

```
class CircleObject extends AxisAlignedBoundingBoxObject implements IDrawable {

  public CircleObject(int x, int y, int r) { super(x, y, r * 2, r * 2); }

  @Override
  public void draw(Graphics2D g2d) {
    g2d.drawOval(this.getX(), this.getY(), this.getWidth(), this.getHeight());
  }
}
```

Listing 5.51—Rectangle Object Class Definition

```
class RectangleObject extends AxisAlignedBoundingBoxObject implements IDrawable {

  public RectangleObject(int x, int y, int w, int h) { super(x, y, w, h); }

  @Override
  public void draw(Graphics2D g2d) {
    g2d.drawRect(this.getX(), this.getY(), this.getWidth(), this.getHeight());
  }
}
```

Example 5.12. A terminal argument parser is a program/function that interprets a series of arguments passed to another program and makes it easier for programmers to determine if a flag is enabled. Without one, many programmers often resort to using a complex series of conditional statements to check for the existence of a flag. Not only is this cumbersome, but it is prone to errors, and neither extendable nor flexible to different arrangements of arguments. In this example we will develop a small terminal argument parser.

First, we need to design a class that represents an "argument" to a program. Arguments, as we described in Chapter **??**, are space-separated string values that we pass to a program executable, which populate the `String[] args` array in the `main` method. In particular, however, we want to specify that an argument is not necessarily the values themselves, but are instead the flags, or instructions, passed to the executable. The simplest version of a flag is one that receives exactly one argument, which we will represent via an abstract `Argument` class. Later on, we want to be able to validate a flag with its given arguments, so the `Argument` class includes an abstract `boolean validate` method, that shall be overridden in all subclasses of `Argument`.

Listing 5.52—Argument Class Definition

```
import java.util.List;
import java.util.Map;

abstract class Argument {

  private String key;

  public Argument(String key) { this.key = key; }

  public String getKey() { return this.key; }

  public abstract boolean validate(Map<String, List<String>> args);
}
```

From here, let's design two types of arguments: one that is optional and one that receives n arguments. Namely, an optional argument is one that is always valid, according to `validate`, because it does not necessarily need to exist. The n-valued argument, on the other hand, requires that the associated passed flag contains exactly n values. For example, if we say that the `--input` flag requires exactly 3 arguments, then if we do not pass exactly three space-separated non-flag values, it fails to validate.

Listing 5.53—Optional Argument Implementation

```
import java.util.List;
import java.util.Map;

class OptionalArgument extends Argument {

  public OptionalArgument(String key) { super(key); }

  @Override
  public boolean validate(Map<String, List<String>> args) { return true; }
}
```

Listing 5.54—Numbered Argument Implementation

```
import java.util.List;
import java.util.Map;

class NumberedArgument extends Argument {

  private final int NUM_REQUIRED_ARGS;

  public NumberedArgument(String key, int n) {
    super(key);
    this.NUM_REQUIRED_ARGS = n;
  }

  @Override
  public boolean validate(Map<String, List<String>> args) {
    if (!args.containsKey(this.getKey())) { return false; }
    else { return args.get(this.getKey()).size() == this.NUM_REQUIRED_ARGS; }
  }
}
```

Now comes the argument parser itself, which receives a string array of argument values, much like `main`, and extracts out the flags and arguments into a `Map<String, List<String>>` where the key represents the flag and the value is a list of the arguments to said flag. We also store a `Set<Argument>` to allow the programmer to designate arguments to the parser. The idea is straightforward: while traversing over `args`, if we encounter a string that begins with a double dash '`--`', it is qualified as a flag and the following arguments, up to another flag, are marked as arguments to the flag. We add these to the respective map as described before, and continue until we run out of elements in the array.

Listing 5.55—Terminal Arguments Parser Implementation

```java
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.HashSet;

class ArgumentParser {

  private Map<String, List<String>> parsedArguments;
  private Set<Argument> arguments;

  public ArgumentParser(String[] args) {
    this.arguments = new HashSet<>();
    this.parsedArguments = new HashMap<>();
    String currKey = null;
    for (String arg : args) {
      if (arg.startsWith("--")) {
        currKey = arg.split("--")[1];
        this.parsedArguments.putIfAbsent(currKey, new ArrayList<>());
      } else if (currKey != null) {
        this.parsedArguments.get(currKey).add(arg);
      }
    }
  }

  public void addArgument(Argument arg) { this.arguments.add(arg); }

  public List<String> getArguments(String key) {
    return this.parsedArguments.containsKey(key) ?
             this.parsedArguments.get(key) :
             null;
  }
}
```

The `parseArguments` method returns whether or not the supplied arguments are valid according to the arguments populated via `addArgument`. Using streams, we verify that, after invoking `validate` on every argument, each separate call returns true, meaning that all arguments are valid and correct. Because it might be useful to return the associated arguments to a flag from a programmer's perspective who uses this parser, we include a `getArguments` method to return the list of arguments passed to a flag.

Listing 5.56—Tester for Terminal Arguments Parser

```java
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.HashSet;
```

```java
class ArgumentParser {

  // ...  other methods not shown.

  /**
   * ...
   * @return
   */
  public boolean parseArguments() {
    return this.arguments.stream()
                         .allMatch(e -> e.validate(this.parsedArguments));
  }
}
```

Example 5.13. Inheritance is a truly powerful programming language construct, and we will now attempt to describe its beauty through the design of a mini-project. Said mini-project will encompass writing a small programming language. Programming language syntax and semantics, collectively, require a lot of knowledge outside the domain and scope of this text, but we will see that, even with our somewhat limited arsenal of tools, we can construct a fairly powerful programming language. Our language will start off as a recreation of the interpreter from our section on interfaces, but contains modifications to make it more flexible.

Programming language syntax is often broken up into the nodes of an *abstract syntax tree*, which at a quick glance is nothing more than a description of the operations of a language. To begin, we need to describe our programming language capabilities. To keep things simple, our language will contain integers, variables, a few arithmetic operators, and conditionals. It's important to note that, because we are glossing over the innards of lexing and parsing, all of our tests will exist in the form of abstract syntax trees. We want an abstract AST node class from which every other AST node inherits. Then, we can design purpose-specific nodes that do what we wish. Every abstract syntax tree has a list of children node. We will also define a `toString` method that will print out the abstract syntax tree in a readable format. Our abstract syntax tree class uses two constructors: one that receives a list of abstract syntax tree nodes, and another that is variadic over the `AstNode` type. We implement two different constructors for convenience purposes during testing.

Additionally, we want our abstract syntax trees to be evaluable. Because "evaluable" describes a behavior of a class, we should throw this into an interface. We want its method, namely `eval`, to return something of type `Lvalue` and receive an environment. For the time being, since we do not know what either an `Lvalue` or an `Environment` is, we will omit the definition of the interface, but include the overridden method inside `AstNode`. Notice, however, that we mark `eval` as `abstract` inside `AstNode`, because it is definitionally impossible to evaluate an `AstNode`, since evaluation behavior is dependent on the subclasses and how they interact.

An `Lvalue` is the left-hand side of the evaluation of an abstract syntax tree. That is, it is the value that a node resolves to after evaluation. As an example, consider an abstract syntax tree that represents a conditional expression. After evaluating the tree, we expect the resulting value to be either the evaluated consequent or the alternative, neither of which are abstract syntax trees themselves. So, we will design a class that encapsulates an abstract syntax tree, and returns the underlying value when prompted. For our programming language, an abstract syntax tree can only resolve to a number or a boolean, since they are the most primitive forms. Because we may need to retrieve the abstract syntax tree of an l-value, we will provide the relevant accessor method. For simplicity, will compare l-values based on their abstract syntax tree string representations.

Listing 5.57—Abstract Syntax Tree Class

```java
import java.util.List;

public abstract class AstNode implements Evaluable {

  private final List<AstNode> CHILDREN;

  public AstNode(List<AstNode> children) {
    this.CHILDREN = children;
  }

  public AstNode(AstNode... children) {
    this(List.of(children));
  }

  @Override
  public abstract Lvalue eval(Environment env);

  public List<AstNode> getChildren() {
    return this.CHILDREN;
  }

  public abstract String toString();
}
```

Listing 5.58—L-value Class

```java
final class Lvalue {

  private final AstNode VALUE;

  public Lvalue(AstNode value) {
    this.VALUE = value;
  }

  public AstNode getAst() {
    return this.VALUE;
  }

  public double getNumberValue() {
    return ((NumberNode) this.VALUE).getValue();
  }

  public boolean getBoolValue() {
    return ((BoolNode) this.VALUE).getValue();
  }

  @Override
  public boolean equals(Object o) {
    if (o instanceof Lvalue) {
      return this.toString().equals(o.toString());
    } else {
      return false;
```

```
    }
  }
}
```

From here, the simplest three abstract syntax tree nodes are VarNode, NumberNode, and BoolNode, corresponding to variables, numbers, and booleans, respectively. Each of these nodes will have a single value, that being the variable name, number, or boolean. The eval methods of the latter two classes return an l-value that wraps themselves, since these values resolve to themselves. The former, that being VarNode, is a little trickier. We must consider what happens when we evaluate a variable in any other programming language. The language looks up the variable identifier in the list of accessible bindings and returns whatever is the corresponding value. This location of bindings is called an environment in the programming languages nomenclature, and generally takes the form of a HashMap data structure. Therefore we can store an instance of a HashMap in our Environment class. The question now is of what type are the keys and values in our map. Fortunately this is simple to determine; the keys are string identifiers, and the values are their corresponding abstract syntax trees. Our environment representation/class is extremely simple and almost seems superfluous, but in due time we will add more functionality to justify its existence over a simple HashMap instance.

Listing 5.59—Environment Class Implementation

```java
import java.util.HashMap;
import java.util.Map;

final class Environment {

  private final Map<String, AstNode> ENV;

  public Environment() { this.ENV = new HashMap<>(); }
}
```

Listing 5.60—Testing Abstract Syntax Tree Tests

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

class AstTest {

  @Test
  void testVarNode() {
    assertEquals("x", new VarNode("x").toString());
  }

  @Test
  void testNumberNode() {
    assertEquals("42", new NumberNode("42").toString());
  }

  @Test
  void testBoolNode() {
    assertEquals("true", new BoolNode("true").toString());
    assertEquals("false", new BoolNode("false").toString());
  }
}
```

Listing 5.61—Variable Node

```java
final class VarNode extends AstNode {

  private final String NAME;

  public VarNode(String name) {
    super();
    this.NAME = name;
  }

  /**
  * Interpret a variable. We look up the variable in the environment and
  * return the value associated with it.
  * @param env The environment in which to interpret the variable.
  * @return The result of the variable lookup after interpretation.
  */
  @Override
  public Lvalue eval(Environment env) {
    String var = this.NAME;
    AstNode res = env.get(var);
    return res.eval(env);
  }

  @Override
  public String toString() { return this.NAME; }
}
```

Listing 5.62—Number Node

```java
final class NumberNode extends AstNode {

  private final double VALUE;

  public NumberNode(String value) {
    super();
    this.VALUE = Double.parseDouble(value);
  }

  public NumberNode(double value) { this(Double.toString(value)); }

  @Override
  public Lvalue eval(Environment env) { return new Lvalue(this); }

  @Override
  public String toString() { return Double.toString(this.VALUE); }
}
```

Listing 5.63—Boolean Node

```java
final class BoolNode extends AstNode {

  private final boolean VALUE;

  public BoolNode(String value) {
    super();
    this.VALUE = Boolean.parseBoolean(value);
  }

  public BoolNode(boolean value) { this(Boolean.toString(value)); }
```

```
  @Override
  public Lvalue eval(Environment env) { return new Lvalue(this); }

  @Override
  public String toString() { return Boolean.toString(this.VALUE); }
}
```

From here, we arrive at primitive operators via `PrimNode`. A primitive operator is one akin to addition, subtraction, and so forth. Two additional primitives that we will support are reading an integer from standard input, and printing one to standard output. Primitive operators receive any number of arguments and whose behavior is handled as a case analysis of the `eval` method.

Listing 5.64—Primitive Operator Node

```
final class PrimNode extends AstNode {

  private final String OP;

  public PrimNode(String op, AstNode... children) {
    super(children);
    this.OP = op;
  }

  /**
   * Interpret a primitive operation.
   * @param env The environment in which to interpret the operation.
   * @return The result of the primitive operation.
   */
  @Override
  public Lvalue eval(Environment env) {
    String op = this.OP;
    // Evaluate each argument before applying the primitive.
    List<Lvalue> operands = this.getChildren().stream()
                                              .map(n -> n.eval(env))
                                              .toList();
    // Each primitive operator is handled differently.
    // For basic things like addition and subtraction, we can simply
    // map the values to their unwrapped types and collect the result.
    return new Lvalue(
      switch (op) {
        case "+" -> new NumberNode(operands.stream()
                                           .map(Lvalue::getNumberValue)
                                           .reduce(0.0, Double::sum));
        case "-" -> new NumberNode(primSub(operands));
        case "*" -> new NumberNode(operands.stream()
                                           .map(Lvalue::getNumberValue)
                                           .reduce(1.0, (acc, c) -> acc * c));
        case "eq?" ->
                new BoolNode(operands.get(0).equals(operands.get(1)));
        default -> null;
      });
  }

  @Override
  public String toString() {
    return String.format("(%s %s)", this.OP, this.getChildren().toString());
  }
}
```

We need a way of binding variables, so we shall take a hint from functional programming languages via the `LetNode` class. The `LetNode` class has three children: a variable name, a value, and a body. The variable name will be a string, with the value and body both being abstract syntax tree nodes. The `LetNode` class will have a `toString` method that will return a string in the form of `(let ([<var> <exp>]) <body>)`. The `eval` method will evaluate the value, extend the environment with the new binding, and then evaluate the body with the extended environment. To do this, we need to understand what environment extension entails. Reconsidering our `Environment` class, we know that it contains a `HashMap` to designate that environments use maps by design. We may be tempted to write a `set` method in our `Environment` class to add an identifier binding to the current environment. Although this works (and will be a necessity in due time), it means that we can only *modify*, or *change*, the environment, which isn't desired. The alternative approach would be to utilize *environment extension*. That is, create a new environment with the old bindings, followed by an insertion of the new binding. Environment extension brings up the issue of variable scope, because different variables are live at different locations in the program. Consider the following program described by the abstract syntax tree:

```
new LetNode("x", new NumberNode(5),
 new LetNode("y", new PrimNode("+", new NumberNode(6), new VarNode("x")),
  new VarNode("y")))
```

Within the inner-most `PrimNode` expression, x does not exist in its environment, but it does exist in an environment defined above its scope. So, each environment is itself a store a map of identifiers to abstract syntax trees, but they also contain another `Environment`. If we are at the "root level" of the program, this environment is set to `null`. Correspondingly, Environment contains two constructors: one that receives a parent environment and another without. As such, we must write the `lookup` method, which finds a variable binding in the current list of bindings and, if it does not exist, recursively looks it up in the parent environment. If we reach the root level environment and the variable does not exist, we return a `null` value. We also override the `toString` method to print out the environment in a readable format.

Listing 5.65—Environment Tester

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

class EnvironmentTester {

  @Test
  void testEnvironment() {
    Environment root = new Environment();
    Environment e1 = env.extend("x", new NumberNode(5));
    Environment e2 = env.extend("y", new NumberNode(6));
    assertAll(
      () -> assertEquals(new NumberNode(5), env.lookup("x")),
      () -> assertEquals(new NumberNode(6), env.lookup("y")),
      () -> assertEquals(null, env.lookup("z")),
      () -> assertEquals(null, env.lookup("x")));
  }
}
```

Listing 5.66—Environment Class

```java
import java.util.HashMap;
import java.util.Map;

final class Environment {

  private final Map<String, AstNode> ENV;
  private final Environment PARENT;

  public Environment() { this(null); }

  public Environment(Environment parent) {
    this.ENV = new HashMap<>();
    this.PARENT = parent;
  }

  /**
   * Looks up a variable in the current environment.
   * @param var the variable name.
   * @return the value bound to the variable, or null if it does not exist.
   */
  @Override
  public AstNode lookup(String var) {
    if (this.ENV.containsKey(var)) { return this.ENV.get(var); }
    else if (this.PARENT != null) { return this.PARENT.ENV.lookup(var); }
    else { return null; }
  }

  /**
   * Extends the current environment to contain a new variable binding.
   * @param var the variable name.
   * @param value the value to bind to the variable.
   * @return a new environment with the new binding.
   */
  public Environment extend(String var, AstNode value) {
    Environment env = new Environment(this);
    env.ENV.put(var, value);
    return env;
  }
}
```

Our modified version of the environment allows us to implement local/let bindings in a way that respects parent environments. As one of the exercises of this chapter demonstrate, environment extension helps us when adding user-defined functions.

Listing 5.67—Let Node

```java
final class LetNode extends AstNode {

  private final String VAR;

  public LetNode(String var, AstNode exp, AstNode body) {
    super(exp, body);
    this.VAR = var;
  }

  /**
   * Interpret a let statement. A new environment is introduced
   * in which the let body is evaluated.
```

```
   * @param env The environment in which to interpret the let binding.
   * @return The result of the let statement.
   */
  @Override
  public Lvalue eval(Environment env) {
    String var = this.VAR;
    AstNode exp = this.getChildren().get(0);
    AstNode body = return this.getChildren().get(1);

    // Interpret the expression and convert it into its AST.
    AstNode newExp = exp.eval(env).getAst();
    Environment e1 = env.extend(var, newExp);
    return body.eval(e1);
  }

  @Override
  public String toString() {
    AstNode e = this.getChildren().get(0);
    AstNode b = this.getChildren().get(1);
    return String.format("(let ([%s %s]) %s"), this.VAR, e.toString(), b.toString
        ());
  }
}
```

Finally we arrive at decision-based nodes. The `IfNode` class represents a conditional expression rather than a statement. Recall the ternary operator; it resolves to a value, unlike Java's `if` statement. The `IfNode` class has three children: a predicate, a consequent, and an alternative. The predicate is an abstract syntax tree that represents a boolean expression, and the consequent and alternative are arbitrary abstract syntax tree nodes. The `IfNode` class will have a `toString` method that will return a string of the form `(if <pred> <conseq> <alt>)`. Its evaluator will evaluate the predicate, and then evaluate either the consequent or alternative depending on the result of the predicate.

Listing 5.68—If Node

```
final class IfNode extends AstNode {

  public IfNode(AstNode predicate, AstNode consequent, AstNode alternative) {
    super(predicate, consequent, alternative);
  }

  /**
  * Interpret an if statement.
  * @param env The environment in which to interpret the if statement.
  * @return The result of the if statement.
  */
  @Override
  public Lvalue eval(Environment env) {
    AstNode pred = this.getChildren().get(0);
    AstNode cons = this.getChildren().get(1);
    AstNode alt = this.getChildren().get(2);

    // Evaluate the predicate, then interpret one way or the other.
    if (pred.eval(env).getBooleanValue()) { return cons.eval(env); }
    else { return alt.eval(env); }
  }

  @Override
  public String toString() {
```

```
    AstNode p = this.getChildren().get(0);
    AstNode c = this.getChildren().get(1);
    AstNode a = this.getChildren().get(2);
    return String.format("(if %s %s %s)", p.toString(), c.toString(), a.toString()
        );
  }
}
```

Finally, at long last, we can write some tests! We will store each test in a class called `InterpTester`, which polymorphically tests the evaluation method of the abstract syntax tree methods. These tests all receive a blank environment representing the global environment. Unfortunately, we still have to write the programs as an abstract syntax tree rather than as a program, but the problems of lexing and parsing are reserved for some other time (or perhaps a separate course altogether).

Listing 5.69—Interpreter Tester

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

class InterpTest {

  @Test
  void interpret() {
    assertAll(
      () -> assertEquals(new Lvalue(new NumberNode("42")),
                         new NumberNode("42").eval(new Environment())),
      () -> assertEquals(new Lvalue(new NumberNode("42")),
                         new LetNode("x", new NumberNode("42"), new VarNode("x"))
                         .eval(new Environment())),
      () -> assertEquals(new Lvalue(new NumberNode("42")),
                         new PrimNode("+", new NumberNode("1"), new NumberNode("41
                             "))
                         .eval(new Environment())),
      () -> assertEquals(new Lvalue(new NumberNode("42")),
                         new LetNode("x",
                          new NumberNode("1"),
                           new LetNode("y",
                            new NumberNode("41"),
                             new PrimNode("+", new VarNode("x"), new VarNode("y"))
                                 ))
                         .eval(new Environment())));
  }
}
```

In general, object-oriented programs with inheritance should be structured as a sequence of specific subclasses that extend an abstract class, as we have done with the different abstract syntax tree node types and the root `AstNode` abstract class.

Chapter Exercises

Exercise 5.1. (⋆⋆)
Design the generic static method `T validateInput(String prompt, String errResp, U extends Predicate<T> p)` that receives a prompt, an error response, and an object that implements the `Predicate` interface to test whether or not the received value, received through standard input, is valid. If the value is invalid according to the predicate, print the error response and re-prompt the user. Otherwise, return the entered value.

Exercise 5.2. (⋆⋆⋆)
A *lazy list* is one that, in theory, produces infinite results! Consider the `ILazyList` interface below:

Listing 5.70

```
interface ILazyList<T> {
  T next();
}
```

When calling `next` on a lazy list, we update the contents of the lazy list and return the next result. We mark this as a generic interface to allow for any desired return type. For instance, below is a lazy list that produces factorial values:[1]

Listing 5.71

```
class FactorialLazyList implements ILazyList<Integer> {

  private int n;
  private int fact;

  public FactorialLazyList() {
    this.n = 1;
    this.fact = 1;
  }

  @Override
  public int next() {
    this.fact *= this.n;
    this.n++;
    return this.fact;
  }
}
```

Testing it with ten calls to `next` yields predictable results.

Listing 5.72—Testing Factorial Lazy List

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

class FactorialLazyListTester {

  @Test
  void testFactorialLazyList() {
    ILazyList<Integer> FS = new FactorialLazyList();
```

[1]We will ignore the intricacies that come with Java's implementation of the `int` datatype. To make this truly infinite, we could use `BigInteger`.

```
    assertAll(
      () -> assertEquals(1, FS.next()),
      () -> assertEquals(2, FS.next()),
      () -> assertEquals(6, FS.next()),
      () -> assertEquals(24, FS.next()),
      () -> assertEquals(120, FS.next()),
      () -> assertEquals(720, FS.next()),
      () -> assertEquals(5040, FS.next()),
      () -> assertEquals(40320, FS.next()),
      () -> assertEquals(362880, FS.next()),
      () -> assertEquals(3628800, FS.next()));
  }
}
```

Design the `FibonacciLazyList` class, which implements `ILazyList<Integer>` and correctly overrides `next` to produce Fibonacci sequence values. You code should *not* use any loops or recursion. Recall that the Fibonacci sequence is defined as $f(n) = f(n-1) + f(n-2)$ for all $n \geq 2$. The base cases are $f(0) = 0$ and $f(1) = 1$.

Listing 5.73—Testing Fibonacci Lazy List

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

class FibonacciLazyListTester {

  @Test
  void testFibonacciLazyList() {
    ILazyList<Integer> FS = new FibonacciLazyList();
    assertAll(
      () -> assertEquals(0, FS.next()),
      () -> assertEquals(1, FS.next()),
      () -> assertEquals(1, FS.next()),
      () -> assertEquals(2, FS.next()),
      () -> assertEquals(3, FS.next()),
      () -> assertEquals(5, FS.next()),
      () -> assertEquals(8, FS.next()),
      () -> assertEquals(13, FS.next()),
      () -> assertEquals(21, FS.next()),
      () -> assertEquals(34, FS.next()));
  }
}
```

Exercise 5.3. (⋆⋆)
Design the `LazyListTake` class. It should receive an `ILazyList` and an integer n denoting how many elements to take, as parameters. Then, write a `List<T> getList()` method, which returns a `List<T>` of n elements from the given lazy list.

Listing 5.74

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

class LazyListTakeTester {

 @Test
 void testLazyListTake() {
  LazyListTake llt1 = new LazyListTake(new FactorialLazyList(), 10);
  LazyListTake llt2 = new LazyListTake(new FibonacciLazyList(), 10);
```

```
    assertAll(
      () -> assertEquals("[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]",
                         llt1.getList().toString()),
      () -> assertEquals("[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]",
                         llt2.getList().toString()));
  }
}
```

Exercise 5.4. (★★)
Java's functional API allows us to pass lambda expressions as arguments to other methods, as well as method references (as we saw in Chapter 3). Design the generic `FunctionalLazyList` class to implement `ILazyList`, whose constructor receives a unary function `Function<T, T> f` and an initial value `T t`. Then, override the `next` method to invoke f on the current element of the lazy list and return the previous. For example, the following test case shows the expected results when creating a lazy list of infinite positive multiples of three.

Listing 5.75

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

class FunctionalLazyListTester {

  @Test
  void testMultiplesOfThreeLazyList() {
    ILazyList<Integer> mtll = new FunctionalLazyList<>(x -> x + 3, 0);
    assertAll(
      () -> assertEquals(0, mtll.next()),
      () -> assertEquals(3, mtll.next()),
      () -> assertEquals(6, mtll.next()),
      () -> assertEquals(9, mtll.next()),
      () -> assertEquals(12, mtll.next()));
  }
}
```

What's awesome about this exercise is that it allows us to define the elements of the lazy list as any arbitrary lambda expression, meaning that we could redefine `FactorialLazyList` and `FibonacciLazyList` in terms of `FunctionLazyList`. We can generate infinitely many ones, squares, triples, or whatever else we desire.

Exercise 5.5. (★★)
Design the generic `CyclicLazyList` class, which implements `ILazyList`, whose constructor is variadic and receives any number of values. Upon calling `next`, the cyclic lazy list should return the first item received from the constructor, then the second, and so forth until reaching the end. After returning all the values, cycle back to the front and continue. For instance, if we invoke `new CyclicLazyList<Integer>(1, 2, 3)`, invoking `.next` five times will produce 1, 2, 3, 1, 2.

Exercise 5.6. (★)
Design the `static <T> Predicate<T> orEquals(Predicate<T> p, T x)` method that, when given a predicate p and an object x, returns a *new* predicate that returns true if its argument x' is equal (using `.equals`) to x or satisfies $p(x)$.

Exercise 5.7. (★★)
Design the `static <T> List<T> predicateOrEquals(List<T> ls, Predicate<T>`

p, BiFunction<T, T, Boolean>, T x) method that, when given a list of values ls, a predicate p, a function f, and a value x that returns the list of values in ls that either satisfy p or are equal according to f. For the purposes of this question, f is a method of two arguments of type T that determines whether or not they are "equal" according to some criteria.

Exercise 5.8. (⋆)
Design the static <T> boolean andMap(List<T> ls, Predicate<T> p) method that returns whether or not all elements of the input list satisfy the given predicate. Use the stream API to solve this problem, but do *not* use the allMatch method, as that method solves the problem we want *you* to solve!

Exercise 5.9. (⋆)
This exercise involves the doubly-linked list we wrote in the chapter. Design the int size() method, which returns the number of elements in the list. You can do this either recursively or with a loop. For better practice, try (and thoroughly test) both implementations.

Exercise 5.10. (⋆⋆)
This exercise involves the doubly-linked list we wrote in the chapter. Design the void set(int i, T v) method, which overwrites/assigns, at index i, the value v. If the provided index is out-of-bounds, do nothing.

Exercise 5.11. (⋆⋆)
This exercise involves the doubly-linked list we wrote in the chapter. Design the void insert(int i, T v) method, which inserts the value v at index i. As an example, if we insert 4 into the list [20, 5, 100, 25] at index 2, the list then becomes [20, 5, 4, 100, 25]. If the provided index is out-of-bounds, do nothing.

Exercise 5.12. (⋆)
This exercise involves the interpreter we wrote in the chapter. Add the "read-number" and "print" primitive operations to the language. The latter is polymorphic, meaning it can print both numbers and booleans.

Exercise 5.13. (⋆⋆)
This exercise involves the interpreter we wrote in the chapter. Functional programming languages, in general, are a composition of expressions, wherein statements are more of an afterthought. To this end, design the BeginNode abstract syntax tree node, which receives a list of abstract syntax trees. At the interpreter level, the BeginNode should evaluate each of the abstract syntax trees in the list, and return the result of the last one.

Exercise 5.14. (⋆⋆)
This exercise involves the interpreter we wrote in the chapter. Variables, in our language, are defined and bound exactly once, namely when they are defined within a let node. Though, in imperative programming, it is often crucial to allow variable reassignments. Design the SetNode class, which receives a variable and an abstract syntax tree, and reassigns the variable to the result of the abstract syntax tree. At the interpreter level, the SetNode should evaluate the abstract syntax tree, and reassign the variable to the result in the current environment (and only the current environment). This means that you'll need to modify the Environment class to allow for variable reassignments. Hint: create a set method in the Environment class.

Exercise 5.15. (⋆⋆)
This exercise involves the interpreter we wrote in the chapter. Recursion is nice and intuitive, for the most part. Unfortunately, it is not always the most efficient way to solve a problem. For example, the Fibonacci sequence, as we saw in Chapter 2, is often defined recursively, but it is much more efficient to define it iteratively (or even with tail recursion). Design the `WhileNode` class, which receives a condition and an abstract syntax tree, and evaluates the abstract syntax tree until the condition is false. At the interpreter level, the `WhileNode` should evaluate the condition, and if it is true, evaluate the abstract syntax tree, and repeat until the condition is false. To test your implementation, you will need to combine the `WhileNode` with both the `SetNode` and `BeginNode` classes.

Exercise 5.16. (⋆⋆⋆)
This exercise involves the interpreter we wrote in the chapter. Having to manually update our case analysis on the primitive operator type is cumbersome and prone to mistakes. A better solution would be to store the operator and its corresponding "handler" method, i.e., the method that receives the operands and does the logic of the operator. We can do this via a map where the keys are the string operators and the values are functional references to the handlers. Unfortunately, Java does not directly support passing methods as parameters, meaning they are not first-class. Conversely, we can make use of Java's functional interfaces to achieve our goal. Namely, the interface will contain one method: `Lvalue apply(List<Lvalue> args, Environment env)`, where `args` is the list of evaluated arguments. We will call the interface `IFunction` and make it generic, with the first type quantified to a list of `Lvalue` instances, and the second type quantified to `Lvalue`. Hopefully, the connection between these quantified types and the signature of `apply` is apparent. Using the below definition of `IFunction`, update `PrimNode` to no longer perform a case analysis in favor of the map. We provide an example of populating the map with the initial operators in a `static` block.

Listing 5.76—Functional Interface for Primitive Operators

```java
@FunctionalInterface
interface IFunction<T, R> {

  R apply(T t);
}
```

Listing 5.77—Primitive Node Class with Function Map

```java
import java.util.Map;
import java.util.HashMap;

class PrimNode extends AstNode {

  private static final Map<String, IFunction<List<Lvalue>, Lvalue>> OPERATORS;

  static {
    OPERATORS = new HashMap<>();
    // Store the primPlus operator in the map.
    OPERATORS.put("+", this.primPlus);
  }

  // Other details omitted.

  @Override
```

```java
  public Lvalue eval(Environment env) {
    // TODO.
  }

  /**
   * Evaluates a plus operator.
   */
  private Lvalue primPlus(List<Lvalue> args, Environment env) {
    int result = 0;
    for (Lvalue lv : args) {
      result += lv.getNumberValue();
    }
    return result;
  }
}
```

Exercise 5.17. (★★★)
This exercise is multi-part and involves the interpreter we wrote in the chapter.

(a) First, design the `ProgramNode` class, which allows the user to define a program as a sequence of statements rather than a single expression.

(b) Design the `DefNode` class, which allows the user to create a global definition. Because we're now working with definitions that do not extend the environment, we should use the set method in environment. When creating a global definition via `DefNode`, we're expressing the idea that, from that point forward, the (root) environment should contain a binding from the identifier to whatever value it binds.

(c) Design the `FuncNode` node. We will consider a function definition as an abstract tree node that begins with `FuncNode`. This node has two parameters to its constructor: a list of parameter (string) identifiers, and a single abstract syntax tree node representing the body of the function. We will only consider functions that return values; void functions do not exist in this language.

(d) Design the `ApplyNode` class, which applies a function to its arguments. You do not need to consider applications in which the first argument is a non-function.

Calling/Invoking a function is perhaps the hardest part of this exercise. Here's the idea, which is synonymous and shared with almost all programming languages:

(i) First, evaluate each argument of the function call. This will result in several l-values, which should be stored in a list.

(ii) Convert these to their AST counterparts.

(iii) We then want to create an environment in which the formal parameters are bound to their arguments. Overload the extend method in `Environment` to now receive a list of string identifiers and a list of (evaluated) AST arguments. Bind each formal to its corresponding AST, and return the extended environment.

(iv) Evaluate the function identifier to get its function definition and convert it to an AST.

(v) Call `eval` on the function body and pass the new (extended) environment.

This seems like a lot of work (because it is), but it means you can write really cool programs, including those that use recursion!

```java
new ProgramNode(
```

```
    new DefNode("!",
      new FuncNode(
        List.of("n"),
        new IfNode(
          new PrimNode("eq?",
            new VarNode("n"),
            new NumberNode(0)),
          new NumberNode(1),
          new PrimNode("*",
            new VarNode("n"),
            new ApplyNode("!",
              new PrimNode("-",
              new VarNode("n"),
              new NumberNode(1))))
        )
      )),
    new PrimNode("print", new ApplyNode("!", new NumberNode(5)))
  )
```

Exercise 5.18. (⋆⋆⋆)
This exercise involves the interpreter we wrote in the chapter. Data structures are a core and fundamental feature of programming languages. A language without them, or at least one to build others on top of, suffers severely in terms of usability. We will implement a *cons*-like data structure for our interpreter. In functional programming, we often use three operations to act on data structures akin to linked lists: *cons*, *first*, and *rest*, to construct a new list, retrieve the first element, and retrieve the rest of the list respectively. We can inductively define a cons list as follows:

```
A ConsList is one of:
 - new ConsList()
 - new ConsList(x, ConsList)
```

Implement the cons data structure into your interpreter. This should involve designing the `ConsNode` class that conforms to the aforementioned data definition. Moreover, you will need to update `PrimNode` to account for the `first` and `rest` primitive operations, as well as an `empty?` predicate, which returns whether or not the cons list is empty. Finally, update the `Lvalue` class to print a stringified representation of a `ConsNode`, which amounts to printing each element, separated by spaces, inside of brackets, e.g., $[l_0, l_1, ..., l_{n-1}]$.

Exercise 5.19. (⋆⋆)
This exercise involves the interpreter we wrote in the chapter. Having to manually type out the abstract syntax tree constructors when writing tests is extremely tedious. Design a *lexer* for the language described by the interpreter. That is, the text is broken up into tokens that are then categorized. For example, `'('` might become `OPEN_PAREN`, `"lambda"` might become `SYMBOL`, `"variable-name"` might become `SYMBOL`, and `123.45` might become `NUMBER`. The output of the lexer is a list of tokens. Part of the trick is to ensure that after reading an open parenthesis, the next token is not grabbed as part of the open parenthesis.

Exercise 5.20. (⋆⋆⋆)
This exercise involves the interpreter we wrote in the chapter. Design a parser for the language described by the interpreter. The idea is to tokenize a raw string, then parse the tokens to create an abstract syntax tree that represents the program. A good starting

point would be to parse *all* parenthesized expressions into what we will call `SExprNode`, then traverse over the tree to "correct" them into their true nodes, e.g., whether they are `IfNode`, `LetNode`, and so forth. Realistically, all programs in our language are, at their core, either primitive values or s-expressions.

Exercise 5.21. (⋆⋆)
This exercise involves the interpreter we wrote in the chapter. The Scheme programming language and its derivatives support *code quotation*, i.e., the ability to convert an evaluable expression into data. As an example, if we evaluate `new QuoteNode(new VarNode("x"))`, we receive a symbol as the output, rather than the evaluated symbol via environment lookup. Add the `QuoteNode` class to your interpreter.

Exercise 5.22. (⋆⋆⋆)
In Chapter 2, we discussed tail recursion and an action performed by some programming languages known as tail-call optimization. We know that we can convert any (tail) recursive algorithm into one that uses a loop, and we described said process in the chapter. There is yet another approach that we can mimic in Java with a bit of trickery and interfaces.

The problem with tail recursion (and recursion in general) in Java is the fact that it does not convert tail calls into iteration, which means the stack quickly overflows with activation records. We can make use of a *trampoline* to force the recursion into iteration through *thunks*. In essence, we have a tail recursive method that returns either a value or makes a tail recursive call, such as the factorial example below.[1] Inside our base case, we invoke the `done` method with the accumulator value. Otherwise, we invoke the `call` method containing a lambda expression of no arguments, whose right-hand side is a recursive call to `factTR`. Functions, or lambda expressions, that receive no arguments are called thunks.

Listing 5.78

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

class FactorialTailRecursiveTester {

  @Test
  void testFactTailRecursiveTrampoline() {
    assertAll(
      () -> assertEquals(BigInteger.valueOf(1),
                         factTailCall(BigInteger.valueOf(0), BigInteger.ONE)),
      () -> assertEquals(BigInteger.valueOf(120),
                         factTailCall(BigInteger.valueOf(5), BigInteger.ONE)),
      () -> assertDoesNotThrow(StackOverflowException.class,
                               () -> factTailCall(BigInteger.valueOf(50000),
                                                  BigInteger.ONE)),
    )
  }
}
```

[1] We omit the driver method to shorten the code, as the important part lies inside the recursive implementation.

Listing 5.79

```
import java.lang.BigInteger;

class FactorialTailRecursive {

  /**
   * Tail-recursive factorial function. Uses BigInteger to avoid number overflow
      and
   * thunks to avoid stack overflow.
   * @param n - the number to compute the factorial of.
   * @param acc - the accumulator.
   * @return a tail call that is either done or not done.
   */
  public static ITailCall<BigInteger> factTailCall(BigInteger n, BigInteger acc) {
    if (n.equals(BigInteger.ZERO)) {
      return TailCallUtils.done(acc);
    } else {
      return TailCallUtils.call(() -> factTailCall(n.subtract(BigInteger.ONE), acc
          .multiply(n)));
    }
  }
}
```

The idea is that we have a helper class and method, namely `invoke`, that continuously applies the thunks, **inside a while loop**, until the computation is done. The trampoline analogy is used because we bounce on the trampoline while invoking thunks and jump off when we are "done."

First, design the generic `ITailCall<T>` interface. It should contain only one (non-default) method: `ITailCall<T> apply()`, which is necessary for the `invoke` method. The remaining methods are all default, meaning they must have a body. Design the `boolean isDone()` method to always return false. Design the `T getValue()` method to simply return `null`. Finally, design the `T invoke()` method that stores a local variable and constantly calls `apply` on itself until it is "done."

Second, design the `TailCallUtils` final class to contain a private constructor (this class will only utilize and define two static methods). The two methods are as follows:

- `public static <T> ITailCall<T> call(ITailCall next)`, which receives and returns the next tail call to apply. This definition should be exactly one line long and as simple as it seems.
- `public static <T> ITailCall<T> done(T val)`, which receives the value to return from the trampoline. We need to create an instance of an interface, which sounds bizarre, but is possible only when we provide an implementation of its methods. So, return a `new ITailCall<>()`, and inside its body, override the `isDone` and `getValue` methods with the correct bodies.

Finally, run the factorial test that we provided earlier in its JUnit suite. It should pass and not stack overflow, hence the inclusion of an `assertDoesNotThrow` call.

Exercise 3.23. (⋆⋆⋆)
Recall the unification exercise from Chapter 3. We can take the idea of unification a step further, which is the basis for almost all logic programming languages such as Prolog. For instance, take the expression `p(X, f(Y))`; attempting to unify this with `p(q(r(x)), f(b(x)))` returns a unification assignment of `X : q(r(x)), Y : b(x)`. It is possible

for a unification to not return any possible assignment. As an example, unifying `p(a, b)` with `p(Y, Y)` returns an empty assignment because it is not possible to unify `a` with `Y`, then unify `b` with `Y`.

Design three classes: `Variable`, `Constant`, and `Predicate`. Each of these should implement the `IUnifiable` interface, which supplies one method: `Assignment unify(IUnifiable u, Assignment as)`. An `Assignment` is simply a mapping of `IUnifiable` objects to `IUnifiable` objects, resembling a map data structure. Variables in this small language will be represented as uppercased letters, whereas constants are lowercase. If two `IUnifiable` objects cannot be unified, then `unify` should return `null`.

Constants are straightforward: constants can only be unified with other constants if they are equivalent. Constants can only be unified with variables if that variable does not have an existing assignment and, if it does, it must be equal to `this` constant. Constants cannot be unified with predicates.

Variables can only be unified with constants if the variable does not have an existing assignment and, if it does, it must be equal to the constant passed as an argument. Variables can only be unified with other variables if at least one is bound to a constant; if they are both bound, then they must be equivalent constants.

Predicates can only be unified with variables if the variable does not have an existing assignment and, if it does, it must be equal to `this` predicate. Predicates can only be unified with predicates if it is possible to successfully unify all of its arguments. E.g., `p(a, z(b), c)` unifies with `p(X, z(Y), Z)` because we return the assignment `X : a, Y : b, Z : c`.

Exercise 5.24. (★★★)
In this series of exercises, you will design several methods that act on very large natural numbers resembling the `BigInteger` class. You cannot use any methods from the class, or the class itself. In this problem you will design several methods that act on very large *natural numbers* resembling the `BigInteger` class. You ***cannot*** use any methods from this class, or the class itself.

(a) Design the `BigNat` class, which has a constructor that receives a string. The `BigNat` class stores an `ArrayList<Integer>` as an instance variable. You will need to convert the given string into said list. Store the digits in reverse order, i.e., the least-significant digit (the ones digit) of the number is the first element of the list.

(b) Override the `String toString()` method to return a string representation of the `BigNat` object.

(c) Override the `BigNat clone()` method that returns a new `BigNat` instance that contains the same number.

(d) Override the `boolean equals(BigNat bn)` method to compare two `BigNat` values for equality.

(e) Implement the `Comparable<BigNat>` interface, and override the `int compareTo(BigNat b1, BigNat b2)` method to return the sign of the result of comparing the given `BigNat` (which we will call b) to `this BigNat` (which we will call a). Namely, if $a < b$, return -1, if $a > b$, return 1, otherwise return 0.

(f) Design the `void add(BigNat bn)` method, which adds a `BigNat` to `this BigNat`. The method should not return anything. Note: this problem is harder than it may look at first glance!

(g) Design the `void sub(BigNat bn)` method, which subtracts a `BigNat` from `this` `BigNat`. If the subtrahend (the right-hand side of the subtraction) is greater than the minuend, the result is zero. Over natural numbers, this is called the *monus* operator.

(h) Design the `void mul(BigNat bn)` method, which multiplies a `BigNat` with `this` `BigNat`. Note: remember how we implement multiplication recursively? You shouldn't use recursion for this problem, but what *is* multiplication? Think about the performance implications of this approach. As we will discuss later, it is $\Theta(n^2)$.

(i) Design the `void div(BigNat bn)` method, which divides a `BigNat` with `this` `BigNat`. If the divisor is greater than the dividend, assign the dividend to be zero. If the divisor is zero, do nothing at all. Otherwise, perform integer division. Note: we can implement division recursively. You shouldn't use recursion for this problem, but what *is* division? Think about the performance implications of this approach. As we will discuss later, it is $\Theta(n^2)$.

Exercise 3.25. (⋆⋆⋆)
Quine's method of truth resolution [van Orman Quine, 1950] is a method of automatically reasoning about the truth of a propositional logic statement (recall the exercise from Chapter 2). The method is as follows:

1. Choose an atom P from the statement. Consider two cases: when P is true and when P is false. Derive the consequences of each case. The rules follow those of the propositional logic connectives.
2. Repeat this process for each sub-statement until there are no more sub-statements, and you have only true or false results. If you have *both* true and false results, the statement is a contingency. If all branches lead to true, the statement is a tautology. If all branches lead to false, the statement is a contradiction.

Design several classes to represent a series of well-formed schemata in propositional logic, namely `CondNode`, `BicondNode`, `NegNode`, `AndNode`, `OrNode`, and `AtomNode`, all of which extend a root `Node` class, similar to our representation of the abstract syntax trees within the ASPL interpreter. Then, design three methods: `boolean isContingency(Node t)`, `boolean isContradiction(Node t)`, and `boolean isTautology(Node t)`, which return whether or not the given statement is a contingency, contradiction, or tautology, respectively. You may assume that the input is a well-formed schema. Note that only one of these methods needs a full-fledged recursive traversal over the data; the other two can be implemented in terms of the first.

6. Exceptions & Data I/O

6.1 Exceptions

Exceptions, at their code, are effect handlers. We use exceptions to identify and respond to events that occur at runtime. Java uses objects to implement an exception type hierarchy, with `Throwable` being the highest class in the chain. Any subclass or instance of `Throwable` can be thrown by Java. We will discuss several different exception types by categorizing them into one of two categories: unchecked versus checked checked exceptions.

Unchecked Exceptions

In general, we handle exceptions at either compile time or runtime. The exceptions, themselves, are thrown at runtime, but certain exceptions must be explicitly handled and referenced by the program. An *unchecked exception* is a form of exception whose behavior is dictated by the runtime system, or is caught by the programmer manually. A convenience factor of unchecked exceptions is that we do not *have* to explicitly state what happens when one is thrown. It should also note that the `RuntimeException` class serves as the superclass of all unchecked exceptions.

Example 6.1. Consider what happens when a program contains code that may or may not invoke a division-by-zero. If the divide-by-zero operation occurs, Java automatically throws an `ArithmeticException` with a relevant explanation of the problem. The exception halts the program at the point thereof, but what's interesting is that we can control the behavior of an unchecked exception via the `try/catch` combination. Inside a `try` block we place the code that might throw the exception. Inside the corresponding `catch` block, we initialize an exception variable to whatever the desired caught exception is, e.g., `ArithmeticException`, and we handle it inside of the block. Let's write a method that does nothing more than divides the sum of two numbers by the third.

Listing 6.1

```
import java.lang.ArithmeticException;

class ArithmeticExceptionExample {
```

```
  public double div(int a, int b, int c) {
    return (a + b) / c;
  }

  public double div2(int a, int b, int c) {
    try {
      return (a + b) / c;
    } catch (ArithmeticException ex) {
      System.err.println("div2: / by zero!");
      return 0;
    }
  }
}
```

We define two versions of `div`, where the first does not perform an explicit check for the exception and the second does. In the latter we print a message to the standard error stream and return zero. The preferable resolution is certainly up to the programmer, but in general it makes more sense to throw the exception and halt program execution, in this instance, rather than propagating a zero up to the caller. Another solution might be to return an `Optional` from the method, but `Optional` is more about compositionality rather than exceptions.

Example 6.2. In the above example, we catch the `ArithmeticException` handled by Java. Though, suppose we have a situation in which *we* want to throw the exception. Because the problem arises from a bad parameter, it would be wise to throw an `IllegalArgumentException`, which designates exactly what its name suggests. We manually check to see if the divisor, namely c, is zero and, if so, we throw a new `IllegalArgumentException`. Because `IllegalArgumentException` is an unchecked exception, the caller needs not to handle nor necessarily know that it may invoke the exception. Should we want to signal that as a hint, in the method signature we can specify that the method potentially throws an `IllegalArgumentException`. As the callee that defines the location of an exception invocation, we *only* throw the exception; it is not our responsibility to control the outcome. We can test a new version of `div` by testing whether or not it throws an exception through the `assertThrows` and `assertDoesNotThrow` assertion methods. The thing is, though, neither `assertThrows` nor `assertDoesNotThrow` are not as simple as they appear on the surface; we need to specify *which* exception the code might throw as a reference to the class definition.[1] Additionally, it must be passed as something that is executable. Fortunately, we have worked with executable constructs before: lambda/anonymous functions! Simply wrap the code that might raise an exception inside a lambda, and it works as expected.

Listing 6.2—Throwing Exceptions Tester

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

class IllegalArgumentExceptionExampleTester {

  @Test
  void testIaeException() {
    assertAll(
      () -> assertThrows(IllegalArgumentException.class, () -> div(5, 3, 0)),
```

[1] To reference a class definition as an object, we access `.class` on the class as if it were a static method.

```
      () -> assertDoesNotThrow(div(5, 3, 1)));
  }
}
```

Listing 6.3—Throwing Exceptions

```
class IllegalArgumentExceptionExample {

  public void div(int a, int b, int c) throws IllegalArgumentException {
    if (c == 0) { throw new IllegalArgumentException("div: / by zero"); }
    else { return (a + b) / c; }
  }
}
```

What if we wanted to call `div` from a separate method and process the exception ourselves? Indeed, this is doable. Should we wish to retrieve the exception message (i.e., the message passed to the exception constructor), we can via the `.getMessage` method, which is helpful for producing custom error messages/responses, or redirecting the message to a different destination.

Listing 6.4—Catching Thrown Exceptions

```
class IllegalArgumentExceptionExample {

  public void div(int a, int b, int c) throws IllegalArgumentException {
    if (c == 0) { throw new IllegalArgumentException("div: / by zero"); }
    else { return (a + b) / c; }
  }

  public static void main(String[] args) {
    try {
      double res = div(2, 3, 0);
    } catch (IllegalArgumentException ex) {
      System.err.printf("main: %s\n" ex.getMessage());
    }
  }
}
```

Example 6.3. Arrays and strings both produce unchecked exceptions, resulting from indexing errors, via `ArrayIndexOutOfBoundsException` and `StringIndexOutOfBounds-Exception` respectively, both of which inherit from the `IndexOutOfBoundsException` class. We imagine that these have both been received, with great frustration, from the readers a indeterminate number of times. Of course, an index out of bounds exception stems from accessing data beyond the permissible bounds of some collection or structure.

Listing 6.5—Out-of-bounds Access Examples

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

class IndexOutOfBoundsExceptionExampleTester {

  @Test
  void testOobException() {
    String ex1 = "String";
    int[] ex2 = new int[]{5, 3, 1, 2, 4, 6};
    assertAll(
```

```
        () -> assertThrows(StringIndexOutOfBoundsException.class, () -> ex1.charAt
             (17)),
        () -> assertThrows(StringIndexOutOfBoundsException.class, () -> ex1.charAt
             (-1)),
        () -> assertDoesNotThrow(() -> ex1.charAt(0)),
        () -> assertDoesNotThrow(() -> ex1.charAt(ex1.length() - 1)),
        () -> assertThrows(ArrayIndexOutOfBoundsException.class, () -> ex2[17]);
        () -> assertThrows(ArrayIndexOutOfBoundsException.class, () -> ex2[-1]);
        () -> assertDoesNotThrow(() -> ex2[0]),
        () -> assertDoesNotThrow(() -> ex2[ex2.length - 1]));
    }
  }
```

Another uncomfortably common unchecked exception that many Java programmers encounter is the `NullPointerException`, most often discovered when referencing an object that has yet to be instantiated.

Example 6.4. Casting an object to a type that it is not results in an unchecked `ClassCastException`. By "to a type that it is not," we mean to say that the object is either not an instance of the type or is not a subtype of the type. Primitive datatypes are not subject to this exception, as they are not objects.[1] All primitive datatypes, except for booleans, can be cast into one another. E.g., `int x = (int) 'A';` is valid, as is `char c = (char) 65;`. On the other hand, `String x = (String) new Integer(5);` is not valid, as `Integer` is not a subclass of `String`. We can treat `List<T>` as an `AbstractList<T>` by performing a runtime cast, e.g., `AbstractList<T> x = (AbstractList<T>) ls;`, where `ls` is defined as being of type `List<T>`.

Example 6.5. Sometimes, a program can encounter a state in which it cannot continue. In this case, we can throw an `IllegalStateException`, designating that the program has reached a point that it should not normally. An example might be attempting to access a closed `Scanner` instance.

```
Scanner in = new Scanner(System.in);
in.close();
String s = in.nextLine(); // Throws IllegalStateException!
```

Checked Exceptions

A checked exception is one that the programmer must explicitly handle. The compiler will not allow the program to compile if the code that may throw the exception is not wrapped in either a `try/catch` block, or the method signature does not specify that it throws the exception. Many checked exceptions arise from I/O operations, such as reading from or to a data source. Considering this, we will discuss checked exceptions in the context of I/O operations in the following (non-sub)section.

User-Defined Exceptions

The programmer can define their own exceptions in terms of other exceptions. Exceptions are nothing more than class definitions, which may be extended and inherited.

Example 6.6. Consider defining the `BadStringInputException` class, which inherits from `RuntimeException`. We might define this as an exception that is thrown when, after reading the user's input, we find that the input is not a "alpha string," i.e., a string

[1] No pun intended.

that contains only letters. We can define a constructor that takes a string as an argument, which serves as the message that is passed to the exception.[1]

Listing 6.6—Bad String Input Exception

```
class BadStringInputException extends RuntimeException {

  public BadStringInputException(String msg) {
    super(String.format("BadStringInputException: %s", msg));
  }
}
```

Then, if we write code that reads a string from the user, we can throw a `BadStringInputException` if the input is a non-alphabetic string. The following code segment uses the `matches` method, which receives a regular expression, and returns whether or not the string satisfies the expression. In particular, `[a-zA-Z]+` states that there must be at least one lowercase or uppercase character.

Listing 6.7—Bad String Input Running Example

```
import java.util.Scanner;

class BadStringInputExceptionExample {

  public static void main(String[] args) {
    Scanner in = new Scanner(System.in);
    String s = in.nextLine();
    if (!s.matches("[a-zA-Z]+")) { throw new BadStringInputException(s); }
  }
}
```

6.2 File I/O

Presumably this section discusses file input and output syntax and semantics. Although this is correct, we will also elaborate on reading data from different sources such as websites and even network connections through sockets.

Primitive I/O Classes

Example 6.7. Let's write a program that reads data from a file and echos it to standard output.

Listing 6.8—Reading File Data Example

```
import java.io.*;

class FileInputStreamExample {

  public static void main(String[] args) {
    FileInputStream fis = null;
    String inputFile = "file1.in";
    try {
```

[1]We note that, in general, creating new exceptions is rarely beneficial, since Java provides a plethora of exception definitions that cover most use cases.

```
      fis = new FileInputStream(inputFile);
      // Read in data byte-by-byte.
      int val = -1;
      while ((val = fis.read()) != -1) { System.out.print(val); }
    } catch (FileNotFoundException ex) {
      System.err.printf("main: could not find %s\n", inputFile);
    } catch (IOException ex) {
      System.err.printf("main: an I/O error occurred: %s\n", ex.getMessage());
    } finally {
      fis.close();
    }
  }
}
```

Recall that in the previous section we mentioned checked exceptions, and deferred the discussion until generalized input and output. Now that we are here, we can refresh our memory and actually put them to use. A checked exception is an exception enforced at compile-time.We emphasize the word enforced because the exception is not handled until runtime, but we must place the code that may throw the checked exception within a `try/catch` block, as we did with the file input stream example. Namely, the `FileInputStream` constructor is defined to potentially throw a `FileNotFoundException`, whereas its `read` method throws a generalized `IOException` if some kind of input error occurs. Since `FileNotFoundException` is a subclass/subexception type of `IOException`, we could omit the distinct catch clause for this exception.

When reading from an input source that is not `System.in`, it is imperative to always close the stream resource. So, after we read the data from our file input stream object `fis`, inside the `finally` block, we should invoke `.close` on the instance, which releases the allocated system resources and deems the file no longer available.[1] Expanding upon the `finally` block a bit more, we will say that it is a segment of code that *always* executes, no matter if the preceding code threw an exception. The finally block is useful for releasing resources, e.g,. opened input streams, that otherwise may not be released. Many programmers often forget to close a resource, and then are left to wonder why a file is either corrupted, overwritten, or some other alternative. To remediate this problem, we can use the *try-with-resources* construct, which autocloses the resource.[2]

Example 6.8. Let's use the try-with-resources block to copy the contents of one file to another. In essence, we will write a program that opens a file input stream and a file output stream, each to separate files. Upon reading one byte from the first, we write that byte to the second.

Listing 6.9—Copying File Data Example with Try-With-Resources

```
import java.io.*;

class FileCopyExample {

  public static void main(String[] args) {
    try (FileInputStream fis = new FileInputStream("file1.in");
         FileOutputStream fos = new FileOutputStream("file1.out")) {
      int val = -1;
      while ((val = fis.read()) != -1) { fos.write(val); }
```

[1]We can check whether an input stream is available via the `.available` method.

[2]Not every resource can be autoclosed; the class of interest must explicitly implement the `AutoCloseable` interface to be wrapped inside a try-with-resources block.

```
    } catch (FileNotFoundException ex) {
      System.err.printf("main: could not find %s\n", inputFile);
    } catch (IOException ex) {
      System.err.printf("main: an I/O error occurred: %s\n", ex.getMessage());
    }
  }
}
```

The file input and output stream classes read data as raw bytes from their source/destination streams. In most circumstances, we probably want to read characters from a data source or to a data destination. To do so, we can instead opt to use the `FileReader` class, which extends `Reader` rather than the `InputStream` class. Namely, `FileReader` is for reading text, whereas `FileInputStream` is for reading raw byte content of a file. Therefore a `FileReader` can read only textual files, i.e., files without an encoding, e.g., `.pdf`, `.docx`, and so forth.

Example 6.9. Using `FileReader`, we will once again write an "echo" program, which reads data from its file source and outputs it to standard output. Of course, we may want to output data to a file, in which case we use the dual to `FileReader`, namely `FileWriter`. In summary, `FileWriter` provides several methods for writing strings and characters to a data destination. In this example we will also write some data to a test file, then examine its output based on the method invocations that we make.

Listing 6.10—Examples of File Reader and Writer

```
import java.io.*;

class FileReaderWriterExamples {

  public static void main(String[] args) {
    try (FileReader fr = new FileReader("file1.in")) {
      int c = -1;
      while ((c = fr.read()) != -1) { System.out.print((char) c); }
    } catch (IOException ex) {
      ex.printStackTrace();
    }

    try (FileWriter fw = new FileWriter("file2.out")) {
      fw.write("Here is a string");
      fw.write("\nHere is another string\n");
      fw.write(9);
      fw.write(71);
      fw.write(33);
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

If we open the `file2.out`, we see that it outputs `"Here is a string"` on one line, followed by `"Here is another string"` on the next line. Then, we might expect it to output the numeric strings `"9"`, `"71"`, and `"33"` all on the same line. The `write` method will coerce (valid) numbers into their ASCII character counterparts, meaning that the file contains the tab character, an uppercase `'G'`, and the exclamation point `'!'`. As we will soon demonstrate, working directly with `FileReader` and `FileWriter` is rarely advantageous.

The problem with the file input and output stream classes, as well as the file reader and writer classes, is that they interact directly with the operating system using low-level operations. Constantly invoking these low-level operations is expensive on the CPU for various reasons, and these classes read/write byte-after-byte of data, which is horribly inefficient. The `BufferedReader` and `BufferedWriter` classes aim to alleviate this problem by instantiating buffers for data. Then, when the buffer is full, the data is flushed to either the source or destination. This way, the program makes fewer operating system-level calls, improving overall program performance. To read from a stream, as suggested, we use `BufferedReader`. Its constructor receives an instance of the `Reader` class, which may be one of several classes. For example, to read from a file, we wrap a `FileReader` inside the constructor of `BufferedReader`. Wrapping a `FileReader` inside a `BufferedReader` allows the buffered reader to interplay (using its optimization techniques) with the file reader, which in turn interacts with the operating system. To output data using buffered I/O, we use the analogous `BufferedWriter` class, which receives a `Writer` instance.

Example 6.10. Using `BufferedReader` and `BufferedWriter`, we will write a program that reads data from a file and outputs it to another file.

Listing 6.11—Buffered Reader and Writer Example

```
import java.io.*;

class BufferedReaderWriterExample {

  public static void main(String[] args) {
    try (BufferedReader br = new BufferedReader(new FileReader("file1.in"));
         BufferedWriter bw = new BufferedWriter(new FileWriter("file1.out"))) {
      String line = null;
      while ((line = br.readLine()) != null) { bw.write(line + "\n"); }
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

The benefits of buffered I/O are not obvious to us as the programmers who use these classes. We can, however, directly compare the timed performance of buffered I/O to non-buffered I/O. The following code shows two implementations of reading the contents of a very large file and echoing them to another. We have two defined methods: `buffered` and `nonbuffered`, which utilize the `BufferedReader/Writer` and `FileInput/OutputStream` classes respectively. Upon testing, we see that the buffered variant takes around three seconds to finish, whereas the nonbuffered version took over four minutes!

Listing 6.12—Buffered versus Non-Buffered I/O Performance

```
import java.io.*;

class PerformanceExamples {

  private static void buffered() {
    try (BufferedReader br = new BufferedReader(new FileReader("huge-2m-file.txt")
         );
         BufferedWriter bw = new BufferedWriter(new FileWriter("bigfile.out"))) {
      int c = -1;
      while ((c = br.read()) != -1) { bw.write(c); }
```

```
    } catch (IOException ex) { ex.printStackTrace(); }
  }

  private static void nonbuffered() {
    try (FileInputStream br = (new FileInputStream("huge-2m-file.txt"));
         FileOutputStream bw = (new FileOutputStream("bigfile.out"))) {
      int c = -1;
      while ((c = br.read()) != -1) { bw.write(c); }
    } catch (IOException ex) { ex.printStackTrace(); }
  }
}
```

The classes that we have explored thus far are primarily for reading/writing either binary or text data. Perhaps we want to output values that are not strictly strings or raw bytes, e.g., integers, doubles, floats, and other primitives datatypes. To do so, we can instantiate a `PrintWriter` instance, which itself receives an instance of the `Writer` class. A concern for some programmers may be that we lose the benefits of buffered I/O, but this is not the case; the constructor for `PrintWriter` wraps the writer object that it receives in an instantiation of a `BufferedWriter` object. Therefore, we do not forgo any performance gains from buffered writing, while gaining the ability to write non-strictly-text data.

Example 6.11. Using `PrintWriter`, let's output some arbitrary constants and formatted strings to a file.

Listing 6.13—Using the `PrintWriter` Class

```
import java.io.*;

class PrintWriterExample {

  public static void main(String[] args) {
    try (PrintWriter pw = new PrintWriter(new FileWriter("file4.out"))) {
      pw.println(Math.PI);
      pw.println(false);
      pw.printf("This is a %s string with %c and %d and %f and %b\n",
                "formatted", '&', 42, Math.E, true);
    } catch (IOException ex) { ex.printStackTrace(); }
  }
}
```

And thus the contents of `file4.out` are, as we might expect:

```
3.141592653589793
false
This is a formatted string with & and 42 and 2.718282 and true
```

We now have methods for reading strings and raw bytes, as well as methods for outputting all primitives and formatted strings to data destinations. We still have one problem: how can we output the representation of an object? For example, take the `BigInteger` class; it has associated instance variables and fields that we also need to store. For this particular class, it might be tempting to store a stringified representation, but this is not an optimal solution because, what if a class has a field that itself is an object? We would need to recursively stringify the object, which is not a good idea. Instead, we can use the `ObjectOutputStream` and `ObjectInputStream` classes, which allow us to *serialize* and *deserialize* objects. Serialization is the process of converting an object into a stream of (transmittable) bytes, whereas deserialization is the opposite process. In summary, when we serialize objects, we save the object itself, alongside any relevant information about the object, e.g., its fields, instance variables, and so forth. Upon deserializing said object, we can restore the object to its original state, initializing its fields. Suppose, on the contrary,

Example 6.12. Let's use `ObjectInput/OutputStream` classes to serialize an object of type `Player`, which has a name, score, health, and array of top scores. To designate that an object can be serialized, it must implement the `Serializable` interface. This interface is a *marker interface*, meaning that it has no methods to implement. Instead, it is a flag that tells the compiler that the class can be serialized. Our example will also demonstrate the idea that classes can contain other class definitions, which is useful for grouping related classes together. The `ObjectStreamExample` class defines the private and static `Player` class as described above. Should we open the `player.out` file, we see that it contains incomprehensible data; this is because the data is intended to be read only by a program.

Listing 6.14—(De-)Serializing the `Player` Class

```java
import java.io.Serializable;

class ObjectStreamExample {

  // ...  other information omitted.

  private static class Player implements Serializable {

    private String name;
    private int score;
    private int health;
    private double[] topScores;

    public Player(String name, int score, int health, double[] topScores) {
      this.name = name;
      this.score = score;
      this.health = health;
      this.topScores = topScores;
    }

    @Override
    public String toString() {
      return String.format("Player[name=%s, score=%d, health=%d, topScores=%s]",
                           name, score, health, Arrays.toString(topScores));
    }
  }
}
```

Suppose, on the contrary, that we store objects as strings in a file. This has two problems: first, as we said before, we would need to recursively serialize all compositional objects of the object that we are serializing. Second, we would need to write a parser to read the stringified object and reinitialize its fields. In essence, we have to reinvent worse versions of preexisting classes.

Example 6.13. In the first chapter we saw how to use the `Scanner` class to read from standard input. Indeed, standard input is a source of data input, but we can wrap any instance of `InputStream` or `File` inside a `Scanner` to take advantage of its helpful data-parsing methods. Let's design a method that reads a series of values that represent `Employee` data for a company. We just saw that we can take advantage of `Serializable` to do this for us, but it is helpful to see how we can also use a `Scanner` to solve a similar problem.

We will say that an `Employee` contains an employee identification number, a first name, a last name, a salary, and whether or not they are full-time staff. Each row in the file contains an `Employee` record.

Listing 6.15—Employee Class Definition

```
class Employee {

  private long employeeId;
  private String firstName;
  private String lastName;
  private double salary;
  private boolean fullTime;

  public Employee(long eid, String f, String l, double s, boolean ft) {
    this.employeeId = eid;
    this.firstName = f;
    this.lastName = l;
    this.salary = s;
    this.fullTime = ft;
  }

  @Override
  public String toString() {
    return String.format("[%d] %s, %s | %.2f | FT?=%b",
                         this.employeeId, this.lastName, this.firstName,
                         this.salary, this.fullTime);
  }
}
```

Our method will return a list of `Employee` instances that has been populated after reading the data from the file. In particular, the `nextLong`, `nextDouble`, `nextBoolean`, and `next` methods will be helpful. The `next` method, whose behavior is not obvious from the name, returns the next sequence of characters prior to a whitespace.

To test, we will create a file containing the following contents:

```
123 John Smith 100000.00 false
456 Jane Doe 75000.00 true
789 Bob Jones 50000.00 false
```

Listing 6.16—Testing the `EmployeeScanner` Class

```
import static Assertions.assertAll;
import static Assertions.assertEquals;
import java.util.List;

class EmployeeScannerTester {

  @Test
  void testReadRecords() {
    List<Employee> emps = EmployeeScanner.readRecords("employees.txt");
    assertAll(
      () -> assertEquals(emps.get(0).toString(),
                         "[123] Smith, John | 100000.00 | FT?=false"),
      () -> assertEquals(emps.get(1).toString(),
                         "[456] Doe, Jane | 75000.00 | FT?=true"),
      () -> assertEquals(emps.get(2).toString(),
                         "[789] Jones, Bob | 50000.00 | FT?=false"));
```

```
  }
}
```

Listing 6.17—Reading Employee Records

```java
import java.util.Scanner;
import java.util.List;
import java.util.ArrayList;
import java.io.File;
import java.io.IOException;

class EmployeeScanner {

  /**
   * Reads in a list of employee records from a given filename.
   * @param fileName name of file.
   * @return list of Employee instances.
   */
  public static List<Employee> readRecords(String fileName) {
    List<Employee> records = new ArrayList<>();

    try (Scanner f = new Scanner(new File(fileName))) {
      while (f.hasNextLine()) {
        long eid = f.nextLong();
        String fname = f.next();
        String lname = f.next();
        double s = f.nextDouble();
        boolean ft = f.nextBoolean();
        records.add(new Employee(eid, fname, lname, s, ft));
      }
    } catch (IOException ex) { ex.printStackTrace(); }

    return records;
  }
}
```

At this point we have seen several methods and classes for reading data from different data sources. Let's now write a few more meaningful programs.

Example 6.14. Let's write a program that reads a file containing integers and outputs, to another file, the even integers. Because our input file has only integers, we can use the `Scanner` class for reading the data and `PrintWriter` to output those even integers. To make the program a bit more interesting, we will read the input file from the terminal arguments, and output the even integers to a file whose name is the same as the input file, but instead with the `.out` extension.

Listing 6.18—Reading and Writing Even Integers

```java
import java.io.*;
import java.util.Scanner;

class EvenIntegers {

  public static void main(String[] args) {
    if (args.length != 1) {
      System.err.println("usage: java EvenIntegers <input-file>");
      System.exit(1);
    }
```

```java
    String inputFile = args[0];
    String outputFile = inputFile.substring(0, inputFile.lastIndexOf('.')) + ".out
         ";

    try (Scanner f = new Scanner(new File(inputFile));
         PrintWriter pw = new PrintWriter(new FileWriter(outputFile))) {
      while (f.hasNextInt()) {
        int val = f.nextInt();
        if (val % 2 == 0) { pw.println(val); }
      }
    } catch (IOException ex) { ex.printStackTrace(); }
  }
}
```

Example 6.15. Let's write a program that returns an array containing the number of lines, words, and characters (including whitespaces but excluding newlines) in a given file. The array indices correspond to those quantities respectively. To simplify the program, words will be considered strings as separated by spaces. For example, if the file contains the following contents:

```
This is a test file.
It contains three lines.
Here is the last line.
```

The returned array should be `[3, 14, 46]`. This way we can write JUnit tests to verify that our program works as intended.

Listing 6.19—Testing the `LineWordCharCounter` Class

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

class LineWordCharCounterTester {

  @Test
  void count() {
    int[] counts = LineWordCharCounter.count("file1.in");
    assertAll(
      () -> assertEquals(counts[0], 3),
      () -> assertEquals(counts[1], 14),
      () -> assertEquals(counts[2], 46));
  }
}
```

Listing 6.20—Counting Lines, Words, and Characters

```java
import java.util.Scanner;
import java.util.File;
import java.io.IOException;

class LineWordCharCounter {

  /**
   * Counts the number of lines, words, and characters in a given file.
   * @param fileName name of file.
   * @return array of counts.
   */
  public static int[] count(String fileName) {
    int[] counts = new int[]{0, 0, 0};
    try (Scanner f = new Scanner(new File(fileName))) {
```

```
      while (f.hasNextLine()) {
        String line = f.nextLine();
        counts[0]++;
        counts[1] += line.split(" ").length;
        counts[2] += line.length();
      }
    } catch (IOException ex) { ex.printStackTrace(); }
    return counts;
  }
}
```

Example 6.16. Going further with terminal arguments, let's write a program that receives multiple file names from the terminal, and outputs a file with all of the data concatenated into one. We will throw an exception if the user passes in files that do not all share the same extension. As an example, should the user input

```
java ConcatenateFiles file1.txt file2.txt file3.txt output-file.txt
```

then the program should output a file `output-file.txt` that contains the contents of `file1.txt`, `file2.txt`, and `file3.txt` in that order.

Listing 6.21—Concatenating Files

```
import java.io.*;
import java.util.Arrays;

class ConcatenateFiles {

  /**
   * Determines whether all files have the same extension.
   * @param files array of file names.
   * @return true if all files have same extension, false otherwise.
   */
  private static boolean sameExtension(String[] files) {
    if (files[0].lastIndexOf('.') == -1) { return false; }
    String extension = files[0].substring(files[0].lastIndexOf('.'));
    for (String file : files) {
      if (file.lastIndexOf(".") == -1 ||
          !file.substring(file.lastIndexOf('.')).equals(extension)) {
        return false;
      }
    }
    return true;
  }

  /**
   * Concatenates the contents of a list of files into a single file.
   * @param files array of file names.
   * @param outputFile name of output file.
   */
  private static void concatenate(String[] files, String outputFile) {
    try (BufferedWriter bw = new BufferedWriter(new FileWriter(outputFile))) {
      for (String file : files) {
        try (BufferedReader br = new BufferedReader(new FileReader(file))) {
          String line = null;
          while ((line = br.readLine()) != null) { bw.write(line + "\n"); }
        }
      }
    } catch (IOException ex) { ex.printStackTrace(); }
  }
```

```
  public static void main(String[] args) {
    if (args.length < 3) {
      System.err.println("usage: java ConcatenateFiles <i-files> <o-file>");
      System.exit(1);
    }

    String[] inputFiles = Arrays.copyOfRange(args, 0, args.length - 1);
    String outputFile = args[args.length - 1];

    if (!sameExtension(inputFiles)) {
      System.err.println("ConcatenateFiles: all files must have same extension");
      System.exit(1);
    }

    concatenate(inputFiles, outputFile);
  }
}
```

6.3 Modern I/O Classes & Methods

Aside from the aforementioned classes for working with files and I/O, Java's later versions provide methods and classes that achieve the same task as those that we might otherwise need to write several lines of code.

Example 6.17. To read the lines from a given file, we might open the file using a `BufferedReader` and `FileReader` object, read the values into some collection, e.g., a list, then process those lines accordingly. This gets repetitive, so it might be a good idea to write a method that does this for us, and is an exercise that we provide to the reader. Java 8 introduced two classes: `Files` and `Path` that work with files and paths respectively. Let's use a handy method from `Files`, namely `readAllLines`, to, as its name implies, read the lines from an input file.

Listing 6.22

```
import java.nio.file.Files;
import java.util.List;

class ReadAllLines {

  public static void main(String[] args) {
    try {
      List<String> lines = Files.readAllLines(Path.of("test.txt"));
      // Some processing with lines...
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

We still need to catch an `IOException` because `readAllFiles` might throw one in the event of some I/O error. What may be slightly disappointing is the fact that we cannot wrap this in a try-with-resources block because `readAllLines` opens and closes the file it receives, resulting in what might appear to be less succinct code. Moreover, the method receives a `Path`, rather than a `String`, which we believe to be an attempt made by Java to prevent the programmer from needing to mess with strings and other input resources directly.

Example 6.18. Unfortunately, `readAllLines` is extremely memory-inefficient, requiring us to store a list of every line in the file. Of course, this seems self-explanatory; why would we not want this in the first place? Consider an extremely large dataset, where the input contains one billion rows. Storing this data directly into running memory is not a particularly viable option, at least at the time of writing this text. A solution is to process each line one at a time, similar to how we work with a `BufferedReader` instance. As the section title suggests, though, there is a better way that incorporates streams into the mix. The `Files` class provides the `lines` method, which returns a stream of the lines in the file. Therefore, appealing to the lazy nature of streams, if we never actually use the data from the stream, nothing happens at all. This is a meaningless exercise, so let's write a method that solves the `1BR` challenge: given a file of data points representing measurements of temperatures in differing locations, return an alphabetized string containing the location and, separated by an equals sign, the minimum, maximum, and average temperatures across all data points for that location.

To start this problem, let's consider our options: we have one billion rows of text in the following format: `"LOCATION;TEMP"`, so storing this in direct memory is a challenge that we will not overcome. Instead, let's create a `Map` that maps location identifiers to `Measurement` objects. A `Measurement` stores a number of occurrences, its minimum, maximum, and total-accrued temperature. Each line we read will either update an existing `Measurement` in the map or insert a new key/value pair.

To start, let's design the skeleton for our method, which we will name `computeTemperatures`, as well as the `Measurement` private class. Moreover, when instantiating a new `Measurement` instance, its current minimum, maximum, and total are all equal to the value on the current line.

Listing 6.23

```java
import java.io.IOException;
import java.nio.file.*;
import java.util.*;

class TemperatureComputer {

  /**
   * Returns a string with the locations and their
   * minimum, maximum, and average temperatures.
   * @param filename - input file with locations and
   * temperature separated by ';'.
   * @return String formatted as aforementioned.
   */
  public static String computeMeasurement(String filename) {
    // TODO.
    return null;
  }

  private static class Measurement {

    private double min, max, total;
    private int noOccurrences;

    public Measurement(double t) {
```

```
      this.numOccurrences = 1;
      this.min = t;
      this.max = t;
      this.total = t;
    }

    /**
     * Adds a temperature to this measurement's total.
     * We update the minimum, maximum, total, and
     * number of occurrences respectively.
     */
    public void add(double t) {
      this.noOccurrences++;
      this.total += t;
      this.min = Math.min(this.min, t);
      this.max = Math.max(this.max, t);
    }
  }
}
```

As stated, using a map is the appropriate data structure, so let's instantiate a `HashMap` due to its quick lookup times.[1] Then, we declare, inside a try-with-resources, a `Stream<String>`, returned by the `lines` method. Once either the stream is fully consumed, the stream is closed, or the program execution finishes the try block, the file is also closed. From the stream, we could use a traditional for-each loop, but let's use stream operations instead. For every line l, we want to split it on the semicolon, retrieve the location and temperature, then update the map as necessary. Because we need to update the state of an object if it exists in the map, we will utilize the `putIfAbsent` method, which returns the associating `Measurement` if the key-to-put already exists.

Lastly, we must conjoin the sorted pairs in the map with commas, which we can do via the `sorted()` and `Collectors.joining()` methods. In addition to this, we added a `toString` method to `Measurement` that returns a formatted string containing the minimum, average, and maximum temperatures floated to one decimal. Due to how trivial this is, we omit it in the listing.

Listing 6.24

```
import java.io.IOException;
import java.util.stream.Stream;
import java.nio.file.*;
import java.util.*;

class TemperatureComputer {

  /**
   * Returns a string with the locations and their
   * minimum, maximum, and average temperatures.
   * @param filename - input file with locations and
   * temperature separated by ';'.
   * @return String formatted as aforementioned.
   */
  public static String computeMeasurement(String filename) {
    Map<String, Measurement> measurementMap = new HashMap<>();
    try (Stream<String> lines = Files.lines(Path.of(filename))) {
      lines.forEach(x -> {
```

[1] In a couple of paragraphs, we will explain this decision instead of using a `TreeMap`.

```
          String[] arr = x.split(";");
          String location = arr[0];
          double temp = Double.parseDouble(arr[1]);
          Measurement ms = measurementMap.putIfAbsent(location, new Measurement(temp
               ));
          if (ms != null) { ms.add(temp); }
        });
      } catch (IOException e) {
        throw new RuntimeException(e);
      }
    return measurementMap.keySet()
                         .stream()
                         .sorted()
                         .map(s -> String.format("%s=%s", s, measurementMap.get(s)))
                         .collect(Collectors.joining(", "));
    }

    // ...  other class not shown.
  }
```

With inputs as large as what we assume them to be, we must make reasonable considerations with our choice of data structure. We could, theoretically, use a `TreeMap` and have the program autosort the measurement map pairs, but this is a performance penalty that is greater than using the `sorted` method as provided by the stream API over the map keys. In our tests, using a `TreeMap` amounted to a forty second performance penalty.

Chapter Exercises

Exercise 6.1. (⋆⋆)
Design the `EchoOdds` class, which reads a file of line-separated integers specified by the user (using standard input), and writes only the odd numbers out to a file of the same name, just with the `.out` extension. If there is a non-number in the file, throw an `InputMismatchException`.

Example Run. If the user types `"file1a.in"` into the running program, and `file1a.in` contains the following:

```
5
100
25
17
2
4
0
-3848
13
```

then `file1a.out` is generated containing the following:

```
5
25
17
13
```

Example Run. If the user types `"file1b.in"` into the running program, and `file1b.in` contains the following:

```
5
100
25
17
THIS_IS_NOT_AN_INTEGER!
4
0
-3848
13
```

then the program does not output a file because it throws an exception.

Exercise 6.2. (⋆⋆)
Design the `Capitalize` class, which reads a file of strings (that are not necessarily line-separated) specified by the user (using standard input), and outputs the capitalized versions of the sentences to a file of the same name, just with the `.out` extension. You may assume that a sentence is a string that is terminated by a period and only a period. This problem is harder than it looks because you need to correctly print the string out to the file. If you use a splitting method, e.g., `.split`, you must remember to reinsert the period in the resulting string. There are many ways to solve this problem!

Example Run. If the user types "`file2a.in`" into the running program, and `file2a.in` contains the following *(note that if you copy and paste this input data, you will need to remove the newline before the "`hopefully`" token):*

```
hi, it's a wonderful day. i am doing great, how are you doing. it's
hopefully fairly obvious as to what you need to do to solve this problem.
this is a sentence on another line.
this sentence should also be capitalized.
```

then `file2a.out` is generated containing the following:

```
Hi, it's a wonderful day. I am doing great, how are you doing. It's
hopefully fairly obvious as to what you need to do to solve this problem.
This is a sentence on another line.
This sentence should also be capitalized.
```

Exercise 6.3. (⋆⋆)
Design the `SpellChecker` class, which reads two files: "`dictionary.txt`" and a file specified by the user (through standard input). The specified flie contains a single sentence that may or may not have misspelled words. Your job is to check each word in the file and determine whether or not they are spelled correctly, according to the dictionary of words. If it is not spelled correctly, wrap it inside brackets `[]`. Output the modified sentences to a file of the same name, just with the `.out` extension instead. You may assume that words are space-separated and that no punctuation exist. Hint: use a `HashSet`! Another hint: words that are different cases are not misspelled; e.g., "`Hello`" is spelled the same as "`hello`"; how can your program check this?

Example Run. Assuming `dictionary.txt` contains a list of words, if the user types "`file3a.in`" into the running program, and `file3a.in` contains the following:

```
Hi hwo are you donig I am dioing jsut fine if I say so mysefl but I
will aslo sya that I am throughlyy misssing puncutiation
```

then `file3a.out` is generated containing the following:

```
Hi [hwo] are you [donig] I am [dioing] [jsut] fine if I say so
[mysefl] but I will [aslo] [sya] that I am [thoroughlyy] [misssing]
[puncutiation]
```

Exercise 6.4. (⋆⋆)
A common use for file input and output is data analysis. Design a class `StatisticsDescriptor` that has the following methods:

- `void read(String fileName)`, which reads in a list of numbers from a file into a collection. These numbers can be integers or floating-point values.
- `double mean()`, which returns the mean of the dataset.
- `double stddev()`, which returns the standard deviation of the dataset.
- `double quantile(double q)`, which receives a quantile value $q \in [0, 1]$ and returns the value such that there are q, as a percentage, values below said value. As an example, if our dataset contains `3, 2, 1, 4, 5, 10, 20`, and we call `quantile(0.30)`, then we return 2.8 to indicate that 30% of the values in the dataset are below 2.8.

- `double median()`, which returns the median, or the middle value, of the dataset.
- `double mode()`, which returns the mode, or the most-frequent value, of the dataset.
- `double range()`, which returns the range, or the difference between the maximum and minimum values of the dataset.
- List<Double> outliers(), which returns the numbers that are outliers of the dataset. We consider a value an outlier if it is greater than three standard deviations away from the mean. Refer to the formula for z-score calculation in the exercises from Chapter 1.
- `void output(String fileName)`, which outputs all of the above statistics to the file specified by the parameter (the order is irrelevant). You should output these as a series of "key-value" pairs separated by an equals sign, e.g., `mean=X`. Put each pair on a separate line.

For all methods (except `read`), if the data has yet to be read, throw an `IllegalStateException`.

Exercise 6.5. (⋆⋆⋆)
A binary relation $\mathcal{R}$ is a subset of the cartesian product of two sets A and B. That is, $\mathcal{R} \subseteq A \times B$ such that $A \times B = \{\langle x, y\rangle \mid x \in A \text{ and } y \in B\}$. There are several ways that we can describe binary relations, including reflexive, symmetric, transitive, antisymmetric, asymmetric, irreflexive, and serial.

Design the generic BinaryRelation<T, U> class to represent a mathematical binary relation. It should store a Set<Pair<String, String>>, where the inner pair is the associated tuples of the set. Its constructor should instantiate the set instance variable.

Then, design the following methods:

- `void addTuple(T x, U y)` receives two values x and y of types T and U respectively, and adds them as a tuple to the underlying set.
- `boolean isReflexive()` returns true if the relation is reflexive. A relation $\mathcal{R}$ is reflexive if, for all $x \in S$, $\langle x, x\rangle \in \mathcal{R}$.
- `boolean isSymmetric()` returns true if the relation is symmetric. A relation $\mathcal{R}$ is symmetric if, for all $x, y \in S$, $\langle x, y\rangle \in \mathcal{R}$ and $\langle y, x\rangle \in \mathcal{R}$.
- `boolean isTransitive()` returns true if the relation is reflexive. A relation $\mathcal{R}$ is transitive if, for all $x, y, z \in S$, if $\langle x, y\rangle \in \mathcal{R}$ and $\langle y, z\rangle \in \mathcal{R}$, then $\langle x, z\rangle \in \mathcal{R}$.
- `boolean isIrreflexive()` returns true if the relation is irreflexive. A relation $\mathcal{R}$ is irreflexive if, for all $x \in S$, $\langle x, x\rangle \notin \mathcal{R}$.
- `boolean isAntisymmetric()` returns true if the relation is antisymmetric. A relation $\mathcal{R}$ is antisymmetric if, for all $x, y \in S$, if $\langle x, y\rangle \in \mathcal{R}$ and $\langle y, x\rangle \in \mathcal{R}$, then $x \neq y$.
- `boolean isAsymmetric()` returns true if the relation is asymmetric. A relation is asymmetric if it is both antisymmetric and irreflexive.
- `boolean isSerial()` returns true if the relation is serial. A relation $\mathcal{R}$ is serial if, for all $x \in S$, there exists a $y \in S$ such that $\langle x, y\rangle \in \mathcal{R}$.

Exercise 6.6. (⋆⋆)
Recall the `Optional` class and its purpose. In this exercise you will reimplement its behavior with the `IMaybe` interface with two subtypes `Just` and `Nothing`, representing the existence and absence of a value, respectively. Design the generic `IMaybe` inter-

face, which contains the following three metohds: `T get()`, `boolean isJust()`, and `boolean isNothing()`. The constructors of these subtypes receive either an object of type `T` or no parameter, depending on whether it is a `Just` or a `Nothing`. Throw an `UnsupportedOperationException` when trying to get the value from an instance of `Nothing`.

Exercise 6.7. (⋆⋆)
Redo the "`Maybe`" exercise, only this time implement it as an abstract class/subclass hierarchy. That is, `Maybe` should be an abstract class containing three abstract methods: `T get()`, `boolean isJust()`, and `boolean isNothing()`. The `Just` and `Nothing` classes should be subclasses of `Maybe` and override these methods accordingly. Do not create constructors for these classes. Instead, create static factory methods `Just.of(T t)` and `Nothing.of()` that return an instance of the appropriate class.

7. Searching and Sorting

7.1 Searching

Linear Search

The *linear search* algorithm is a sequential searching algorithm. That is, we check element-by-element to determine if the element we are looking for is in the list. If the element is in the list, we return the index of the element. If the element is not in the list, we (generally) return -1. Linear search is nonsensical for non-constant-time access data structures, such as linked lists, because the whole point of linear search is to retrieve an element at its index quickly, then determine if it is the element we are searching.

Standard Recursive Linear Search

We might think that a standard recursive linear search works, and indeed it is possible to write such an algorithm, but in Java it is almost nonsensical to do so. Consider the possible base cases: if the list is empty, what do we return? As described earlier, we might return -1, but think about what happens when the recursive calls unwind. If we add one to the result of the recursive calls when searching, then if the element is not in the list, the returned value will always be the length of the list minus one. Should we find the desired element, we might return zero, which works as expected. It's only in the cases when we do not find the element that we run into trouble. There are two possible solutions: throw an exception if the element is not present, or simply do not use a standard recursive linear search.[1] We might also think to use `Optional`, but this is not a good idea either, since we would need to check the result of the recursive calls, which detracts from the simplicity of the algorithm.

Another reason why the standard-recursive version is suboptimal is because we have no way of passing the index-to-check; we instead must create sublists of the original list, which is horribly inefficient (at least in Java).

[1]Some languages, e.g., Scheme, use *continuations* to represent "jump-out" states; allowing the program to forgo unwinding the call stack when returning -1. Since Java does not support continuations by defauilt, we cannot use this methodology.

First, let's create an interface for designing linear search algorithms. Recall when we stated that the data structure should guarantee constant-access times. The `List` interface does not provide this promise, so we will instead opt for a generic `AbstractList<V>`, where `V` is any comparable type. The method provided by the interface should receive the list and the element to search.

Listing 7.1—Testing the Linear Search Algorithms

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

import java.util.AbstractList;
import java.util.NoSuchElementException;

class ILinearSearchTest {

  private static final AbstractList<Integer> LS1
    = new ArrayList<>(List.of(78, 43, 22, 101, 29, 34, 23, 12, 33));
  private static final AbstractList<Integer> LS2
    = new ArrayList<>(List.of(1,2,3,4,5,6,7,8,9,10,11,12));

  @Test
  void testSrls() {
    ILinearSearch<Integer> ls = new StandardRecursiveLinearSearch<>();
    assertAll(
      () -> assertEquals(2, ls.linearSearch(LS1, 22)),
      () -> assertThrows(NoSuchElementException.class,
                         () -> ls.linearSearch(LS1, 102)),
      () -> assertEquals(8, ls.linearSearch(LS1, 33)),
      () -> assertEquals(0, ls.linearSearch(LS1, 78)),

      () -> assertEquals(2, ls.linearSearch(LS2, 3)),
      () -> assertThrows(NoSuchElementException.class,
                         () -> ls.linearSearch(LS2, 13)),
      () -> assertEquals(8, ls.linearSearch(LS2, 9)),
      () -> assertEquals(0, ls.linearSearch(LS2, 1)));
  }

  @Test
  void testTrls() {
    ILinearSearch<Integer> ls = new TailRecursiveLinearSearch<>();
    assertAll(
      () -> assertEquals(2, ls.linearSearch(LS1, 22)),
      () -> assertThrows(-1, ls.linearSearch(LS1, 102)),
      () -> assertEquals(8, ls.linearSearch(LS1, 33)),
      () -> assertEquals(0, ls.linearSearch(LS1, 78)),

      () -> assertEquals(2, ls.linearSearch(LS2, 3)),
      () -> assertThrows(-1, ls.linearSearch(LS2, 13)),
      () -> assertEquals(8, ls.linearSearch(LS2, 9)),
      () -> assertEquals(0, ls.linearSearch(LS2, 1)));
  }

  @Test
  void testLLs() {
    ILinearSearch<Integer> ls = new LoopLinearSearch<>();
    assertAll(
      () -> assertEquals(2, ls.linearSearch(LS1, 22)),
      () -> assertThrows(-1, ls.linearSearch(LS1, 102)),
```

```
        () -> assertEquals(8, ls.linearSearch(LS1, 33)),
        () -> assertEquals(0, ls.linearSearch(LS1, 78)),

        () -> assertEquals(2, ls.linearSearch(LS2, 3)),
        () -> assertThrows(-1, ls.linearSearch(LS2, 13)),
        () -> assertEquals(8, ls.linearSearch(LS2, 9)),
        () -> assertEquals(0, ls.linearSearch(LS2, 1)));
  }
}
```

Listing 7.2—Linear Search Interface

```java
import java.lang.Comparable;
import java.util.AbstractList;

interface ILinearSearch<V extends Comparable<V>> {

  int linearSearch(AbstractList<V> ls, V v);
}
```

Listing 7.3—Standard-Recursive Linear Search with Exceptions

```java
import java.lang.Comparable;
import java.util.AbstractList;
import java.util.NoSuchElementException;

class StandardRecursiveLinearSearch<V extends Comparable<V>>
                                    implements ILinearSearch<V> {

  @Override
  public int linearSearch(AbstractList<V> ls, V v) {
    if (ls.isEmpty()) { throw new NoSuchElementException(); }
    else if (ls.get(0).equals(v)) { return 0; }
    else {
      return 1 + linearSearch((AbstractList<V>) ls.subList(1, ls.size()), v);
    }
  }
}
```

As demonstrated, we must handle the no-element case as an exception, which would otherwise be left as a decision to the programmer that calls this version of the `linearSearch` algorithm.

Tail Recursive Linear Search

The tail recursive linear search is much more intuitive to understand compared to its standard recursive counterpart—we use an accumulator to keep track of the current index, and if we reach the end of the list, a -1 is returned. If we find the element v, we return the index, which does not unwind the stack since there is no work remaining after each tail call, making this extremely efficient (again, at least in comparison to the standard recursive version).

Listing 7.4—Tail-Recursive Linear Search

```java
import java.lang.Comparable;
import java.util.AbstractList;

class TailRecursiveLinearSearch<V extends Comparable<V>>
```

```
                              implements ILinearSearch<V> {
  @Override
  public int linearSearch(AbstractList<V> ls, V v) {
    return this.linearSearchHelper(ls, v, 0);
  }

  private int linearSearchHelper(AbstractList<V> ls, V v, int idx) {
    if (idx == ls.size()) { return -1; }
    else if (ls.get(idx).equals(v)) { return idx; }
    else { return linearSearchHelper(ls, v, idx + 1); }
  }
}
```

Loop Linear Search

Even though the tail recursive linear search is relatively straightforward to understand, almost all linear search implementations prefer the iterative version, or one that uses a loop, which we will now show. As an exercise, the readers should go through the process described in Chapter 2 to convert the tail recursive method into an iterative method. We do not do so here because we know the upper-bound of a linear search, so even though a `while` loop is correct, it offers no advantages over a `for` loop.

Listing 7.5—Iterative Linear Search

```
import java.lang.Comparable;
import java.util.AbstractList;

class LoopLinearSearch<V extends Comparable<V>> implements ILinearSearch<V> {

  @Override
  public int linearSearch(AbstractList<V> ls, V v) {
    int idx = 0;
    for (int i = 0; i < ls.size(); i++) {
      if (ls.get(i).equals(v)) { return i; }
    }
    return -1;
  }
}
```

Binary Search

Binary search is the alternative to linear search, and fortunately proves to be significantly faster, but with a catch: the data must be sorted in order to correctly use a binary searching algorithm. Here's how it works: we check the middle element e of the list against our target value k and, if they are equal, we return the middle element index. If $e < k$, we know that k is greater than all elements to the left of e because the data is in sorted order. Therefore, we can check exclusively on the right-half. This idea applies to the left-half as well; if $e > k$, then k must be less than all elements to the right of the middle.

Binary search makes even less sense to design as a standard-recursive algorithm because of the fact that we have to search separate partitions of the list. So, we will only design tail-recursive and loop versions. The tail-recursive variant is extremely simple and directly follows from the English description of the algorithm: we keep track of the indices to search between l and h, assuming $l \leq h$. Given this assumption we compute the middle element index as $l + (h - l)/2$, and recursively update l/h as necessary to change the bounds of the search “zone.” If $l > h$, the bounds have crossed, meaning the search element does not exist in the list.

When we state that binary search is faster than linear search, this is of course relative to the problem context; if we want to search an unsorted list exactly once, then taking the time to sort the data, then run binary search, it is not optimal. Repeatedly searching for elements in a list containing many values should be done with first sorting the list, then binary searching. As an example, if we were to use linear search on a sorted list containing half a billion elements for an element that is not present, we must check all half a billion values. Compare this to binary search, which is logarithmic in the number of elements. So, taking the base two logarithm of our input size places an upper-bound of twenty-nine comparisons (after rounding). To say that this is a substantial performance increase is underselling it to the max.

Listing 7.6—Testing the Binary Search Algorithms

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

import java.util.AbstractList;

class IBinarySearchTest {

  private static final AbstractList<Integer> LS1
    = new ArrayList<>(List.of(12, 22, 23, 29, 33, 34, 43, 78, 101));
  private static final AbstractList<Integer> LS2
    = new ArrayList<>(List.of(1,2,3,4,5,6,7,8,9,10,11,12));

  @Test
  void testTrbs() {
    IBinarySearch<Integer> ls = new TailRecursiveBinarySearch<>();
    assertAll(
      () -> assertEquals(1, ls.binarySearch(LS1, 22)),
      () -> assertEquals(-1, ls.binarySearch(LS1, 102)),
      () -> assertEquals(4, ls.binarySearch(LS1, 33)),
      () -> assertEquals(7, ls.binarySearch(LS1, 78)),

      () -> assertEquals(2, ls.binarySearch(LS2, 3)),
      () -> assertEquals(-1, ls.binarySearch(LS2, 13)),
      () -> assertEquals(8, ls.binarySearch(LS2, 9)),
      () -> assertEquals(0, ls.binarySearch(LS2, 1)));
  }

  @Test
  void testLbs() {
    IBinarySearch<Integer> ls = new LoopBinarySearch<>();
    assertAll(
      () -> assertEquals(1, ls.binarySearch(LS1, 22)),
      () -> assertEquals(-1, ls.binarySearch(LS1, 102)),
      () -> assertEquals(4, ls.binarySearch(LS1, 33)),
      () -> assertEquals(7, ls.binarySearch(LS1, 78)),

      () -> assertEquals(2, ls.binarySearch(LS2, 3)),
      () -> assertEquals(-1, ls.binarySearch(LS2, 13)),
      () -> assertEquals(8, ls.binarySearch(LS2, 9)),
      () -> assertEquals(0, ls.binarySearch(LS2, 1)));
  }
}
```

Listing 7.7—Binary Search Interface

```
import java.lang.Comparable;
import java.util.AbstractList;

interface IBinarySearch<V extends Comparable<V>> {

  int binarySearch(AbstractList<V> ls, V v);
}
```

Tail Recursive Binary Search

Listing 7.8—Tail-Recursive Binary Search

```
import java.lang.Comparable;
import java.util.AbstractList;

class TailRecursiveBinarySearch<V extends Comparable<V>>
                                implements IBinarySearch<V> {

  @Override
  public int binarySearch(AbstractList<V> ls, V v) {
    return binarySearchTRHelper(ls, v, 0, ls.size() - 1);
  }

  private int binarySearchTRHelper(AbstractList<V> ls, V v, int low, int high) {
    if (low > high) { return -1; }
    else {
      int mid = low + (high - low) / 2;
      if (ls.get(mid).compareTo(v) > 0) {
        return binarySearchTRHelper(ls, v, low, mid - 1);
      } else if (ls.get(mid).compareTo(v) < 0) {
        return binarySearchTRHelper(ls, v, mid + 1, high);
      } else {
        return mid;
      }
    }
  }
}
```

Loop Binary Search

Listing 7.9—Iterative Binary Search

```
import java.lang.Comparable;
import java.util.AbstractList;

class LoopBinarySearch<V extends Comparable<V>> implements IBinarySearch<V> {

  @Override
  public int binarySearch(AbstractList<V> ls, V v) {
    int low = 0;
    int high = ls.size() - 1;
    while (low <= high) {
      int mid = low + (high - low) / 2;
      if (ls.get(mid).compareTo(v) > 0) { high = mid - 1; }
      else if (ls.get(mid).compareTo(v) < 0) { low = mid + 1; }
      else { return mid; }
```

```
    }
    return -1;
  }
}
```

Using Comparators for Searching

Recall the use of comparators from 3 when discussing priority queues. Our implementations of linear and binary search, at the moment, require the generic class we parameterize over to be `Comparable`, which can be slightly limiting on the types of classes we can search through, because it is not feasible to modify a class that already exists in the Java library. Plus, having to go back and write a definition of `compareTo` in a Java source file is cumbersome. The solution to this problem is to define a custom `Comparator` object and pass that to the search methods. Namely, we can add a second method to the binary search generic interface that receives an instance of `Comparator`, which is then internally utilized by our search algorithms.

Example 7.1. Let's amend our definition of `IBinarySearch` to also include a method that receives the `Comparator` object. Note that we need to specify in the definition of the comparator that the type it receives should be a superclass of the list element type. Therefore we will use the dual to `extends`: namely `super`, in the parameterized type. Note that we do not necessarily care about the type variable of this element, so we can use a wildcard '?' instead of using another letter from the alphabet. With this modification, however, it no longer makes as much sense to quantify that `V` extends `Comparable<V>`, because the latter method does not require its type to implement that interface. So, let's remove it from the interface signature and move it down to the type quantification.

Listing 7.10—Adding a Comparator-Based Binary Search

```java
import java.lang.Comparable;
import java.util.Comparator;
import java.util.AbstractList;

interface IBinarySearch {

  <V extends Comparable<V>> int binarySearch(AbstractList<V> ls, V v);

  <V> int binarySearch(AbstractList<V> ls, V v, Comparator<? super V> c);
}
```

Now, of course, we do not want to have to rewrite the entire definition of `binarySearch` just to make use of a different comparator, and indeed, we do not have to do so. All we must do is move the logic of the search into the version that receives a comparator and update the one that operates over comparable elements. This means that we need to instantiate a `Comparator` that uses the `compareTo` method from the elements, which is trivial to do via a lambda expression. We also must change the definition from using `compareTo`, which comes from `Comparable` to use `compare` from the `c` object.

Listing 7.11—Testing the Comparator-Based Binary Search

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

import java.util.AbstractList;
```

```java
import java.util.Comparator;

class IBinarySearchTester {

  // Other testing methods omitted.

  @Test
  void testBinarySearchComparator() {
    IBinarySearch<Integer> search = new TailRecursiveBinarySearch<>();
    AbstractList<Integer> ls = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
    Comparator<Integer> c = (o1, o2) -> o1.compareTo(o2);
    assertAll(
      () -> assertEquals(0, search.binarySearch(ls, 1, c));
      () -> assertEquals(9, search.binarySearch(ls, 10, c));
      () -> assertEquals(4, search.binarySearch(ls, 5, c));
      () -> assertEquals(-1, search.binarySearch(ls, 11, c)));
  }
}
```

Listing 7.12—Comparator-Based Tail Recursive Binary Search

```java
import java.lang.Comparable;
import java.util.AbstractList;
import java.util.Comparator;

class TailRecursiveBinarySearch implements IBinarySearch {

  @Override
  public <V extends Comparable<V>> int binarySearch(AbstractList<V> ls, V v) {
    return binarySearchTRHelper(ls, v, 0, ls.size() - 1,
                                (o1, o2) -> o1.compareTo(o2));
  }

  @Override
  public <V> int binarySearch(AbstractList<V> ls, V v, Comparator<? super V> c) {
    return binarySearchTRHelper(ls, v, 0, ls.size() - 1, c);
  }

  private <V> int binarySearchTRHelper(AbstractList<V> ls, V v, int low, int high,
                                       Comparator<? super V> c) {
    if (low > high) { return -1; }
    else {
      int mid = low + (high - low) / 2;
      if (c.compare(ls.get(mid), v) > 0) {
        return binarySearchTRHelper(ls, v, low, mid - 1, c);
      } else if (c.compare(ls.get(mid), v) < 0) {
        return binarySearchTRHelper(ls, v, mid + 1, high, c);
      } else {
        return mid;
      }
    }
  }
}
```

We leave changing the loop variant to use comparators as an exercise. It does not make sense to update `ILinearSearch` because linear search uses `.equals` for comparing objects rather than `.compareTo` or `.compare`.

7.2 Sorting

In this section we will begin our discussion on the analysis and implementation of sorting algorithms. Each algorithm contains two variants: a functional and in-place variant. The functional variant will return a new list that is sorted, while the in-place variant will sort the list in-place. The functional variant is, in principle, easier to implement, but the in-place variant is more efficient in terms of memory usage. Moreover, all lists that are parameters to the sorting algorithms are assumed to be constant-access lists. Accordingly we specify that the input list extends `AbstractList` class, which guarantees our presumption.

For each algorithm, we will assume the same three lists are declared and properly instantiated within the respective unit testing files. To conserve space, we will list their values below only once.

```
AbstractList<Integer> LS1 =
  new ArrayList<>(List.of(5, 4, 2, 1, 3));
AbstractList<Integer> LS2 =
  new ArrayList<>(List.of());
AbstractList<Integer> LS3 =
  new ArrayList<>(List.of(10, 8, 6, 7, 2, 10, 3, 3, 3, 10));
```

Insertion Sort

Our sorting adventure begins with the *insertion sort*. Insertion sort, in general, works by taking an element from the unsorted list and inserting it into the correct position in a sorted list. We can implement insertion sort in two ways: functionally and in-place. The functional variant will return a new list that is sorted, while the in-place variant will sort the list in-place. The functional variant is, in principle, easier to implement, but the in-place variant is more efficient in terms of memory usage. Figure 7.1 illustrates the in-place insertion sort algorithm; each iteration of the outer loop is represented by a row in the figure, with the red underbars representing the now-sorted sublist.

Listing 7.13—Insertion Sort Tester

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

class InsertionSortTester {

  @Test
  void fInsSort() {
    IInsertionSort<Integer> ss = new FunctionalInsertionSort<>();
    assertAll(
      () -> assertEquals(List.of(1, 2, 3, 4, 5), ss.insertionSort(LS1)),
      () -> assertEquals(List.of(), ss.insertionSort(LS2)),
      () -> assertEquals(List.of(2, 3, 3, 3, 6, 7, 8, 10, 10, 10),
                         ss.insertionSort(LS3)));
  }

  @Test
  void ipInsSort() {
    IInsertionSort<Integer> is = new InPlaceInsertionSort<>();
    assertAll(
      () -> is.insertionSort(LS1),
```

```
        () -> assertEquals(List.of(1, 2, 3, 4, 5), LS1),
        () -> is.insertionSort(LS2),
        () -> assertEquals(List.of(), LS2),
        () -> is.insertionSort(LS3),
        () -> assertEquals(List.of(2, 3, 3, 3, 6, 7, 8, 10, 10, 10), LS3));
  }
}
```

Listing 7.14—Insertion Sort Generic Interface

```
import java.util.AbstractList;

interface IInsertionSort<V extends Comparable<V>> {
  AbstractList<V> insertionSort(AbstractList<V> ls);
}
```

Functional Insertion Sort

The functional insertion sort is a recursive sorting algorithm; it sorts the list by recursively sorting its tail (i.e., the list without the first element)

Listing 7.15—Functional Insertion Sort Implementation

```
import java.lang.Comparable;
import java.util.AbstractList;
import java.util.ArrayList;

class FunctionalInsertionSort<V extends Comparable<V>>
                              implements IInsertionSort<V> {

  @Override
  public AbstractList<V> insertionSort(AbstractList<V> ls) {
    if (ls.isEmpty()) { return new ArrayList<>(); }
    else {
      return insert(ls.get(0),
                    insertionSort((AbstractList<V>) ls.subList(1, ls.size())));
    }
  }

  /**
   * Inserts an element into a sorted list of values.
   * @param val value to insert.
   * @param sortedRest a sorted sublist.
   * @return the sorted sublist with the new value inserted.
   */
  private AbstractList<V> insert(V val, AbstractList<V> sortedRest) {
    if (sortedRest.isEmpty()) {
      ArrayList<V> ls = new ArrayList<>();
      ls.add(val);
      return ls;
    } else if (val.compareTo(sortedRest.get(0)) < 0) {
      ArrayList<V> ls = new ArrayList<>();
      ls.add(val);
      ls.addAll(sortedRest);
      return ls;
    } else {
      ArrayList<V> ls = new ArrayList<>();
      ls.add(sortedRest.get(0));
      ls.addAll(insert(val, (AbstractList<V>)
                        sortedRest.subList(1, sortedRest.size())));
```

```
      return ls;
    }
  }
}
```

In-place Insertion Sort

Listing 7.16—In-place Insertion Sort Implementation

```
import java.lang.Comparable;
import java.util.AbstractList;
import java.util.Collections;

class InPlaceInsertionSort<V extends Comparable<V>> implements IInsertionSort<V> {

  @Override
  public AbstractList<V> insertionSort(AbstractList<V> ls) {
    for (int i = 1; i < ls.size(); i++) {
      V curr = ls.get(i);
      int j = i - 1;
      while (j >= 0 && ls.get(j).compareTo(curr) > 0) {
        Collections.swap(ls, j+1, j);
        j--;
      }
    }
    return ls;
  }
}
```

Selection Sort

The selection sort is the next sorting algorithm that we will analyze. It works by first searching for the smallest element in the list, then swapping it with the first element. Then we search for the second smallest element, and swap it with the second element. We continue this process until the list is sorted. Being that we always search the entire list for the smallest element, this is a horrendously slow sorting algorithm and should be avoided in favor of faster approaches. Nevertheless, we show both the functional and in-place versions. Figure **??** illustrates the in-place selection sort algorithm. We admit that the figure is a bit misleading since it gives the false impression that it requires the same number of traversals as insertion sort, but the figures do not represent the number of comparisons made in between each element.

Listing 7.17—Selection Sort Tester

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

import java.util.List;

class SelectionSortTester {

  @Test
  void fSelSort() {
    ISelectionSort<Integer> ss = new FunctionalSelectionSort<>();
    assertAll(
      () -> assertEquals(List.of(1, 2, 3, 4, 5), ss.selectionSort(LS1)),
```

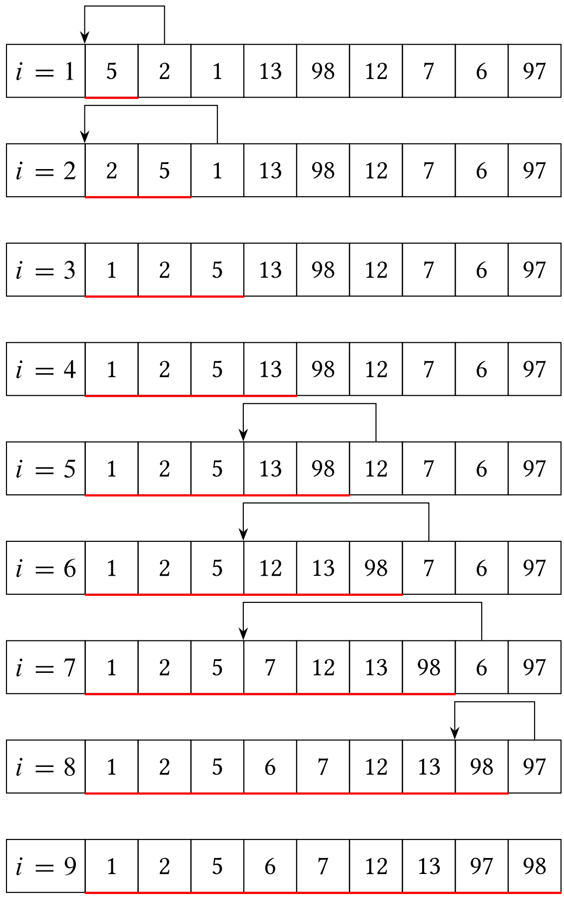


Figure 7.1: In-Place Insertion Sort Illustration

```
      () -> assertEquals(List.of(), ss.selectionSort(LS2)),
      () -> assertEquals(List.of(2, 3, 3, 3, 6, 7, 8, 10, 10, 10),
                         ss.selectionSort(LS3)));
  }

  @Test
  void ipqsTester() {
    ISelectionSort<Integer> ss = new InPlaceSelectionSort<>();
    assertAll(
      () -> ss.selectionSort(LS1),
      () -> assertEquals(List.of(1, 2, 3, 4, 5), LS1),
      () -> ss.selectionSort(LS2),
      () -> assertEquals(List.of(), LS2),
      () -> ss.selectionSort(LS3),
      () -> assertEquals(List.of(2, 3, 3, 3, 6, 7, 8, 10, 10, 10), LS3));
  }
}
```

Listing 7.18—Insertion Sort Generic Interface

```java
import java.lang.Comparable;
import java.util.AbstractList;

interface ISelectionSort<V extends Comparable<V>> {
  AbstractList<V> selectionSort(AbstractList<V> ls);
}
```

Functional Selection Sort

Listing 7.19—Functional Selection Sort Def.

```java
import java.util.*;

class FunctionalSelectionSort<V extends Comparable<V>>
                              implements ISelectionSort<V> {

  @Override
  public AbstractList<V> selectionSort(AbstractList<V> ls) {
    if (ls.isEmpty() || ls.size() == 1) { return ls; }
    else {
      // Remember that min returns an Optional, but we know it is non-empty.
      int minIdx = IntStream.range(0, ls.size())
                            .boxed()
                            .min((i1, i2) -> ls.get(i1).compareTo(ls.get(i2)))
                            .get();

      // Swap the minimum element with the first element.
      Collections.swap(ls, 0, minIdx);

      // Sort the rest of the list (excluding the first element).
      AbstractList<V> sortedRest = selectionSort(
                                    new ArrayList<>(ls.subList(1, ls.size())));

      // Construct the final sorted list.
      AbstractList<V> sortedList = new ArrayList<>();
      sortedList.add(ls.get(0));
      sortedList.addAll(sortedRest);
      return sortedList;
    }
  }
}
```

In-place Selection Sort

Listing 7.20—In-place Selection Sort Implementation

```java
import java.util.*;

class InPlaceSelectionSort<V extends Comparable<V>> implements ISelectionSort<V> {

  @Override
  public AbstractList<V> selectionSort(AbstractList<V> ls) {
    for (int i = 0; i < ls.size(); i++) {
      V min = ls.get(i);
      int minIdx = 0;
      boolean needToSwap = false;
```

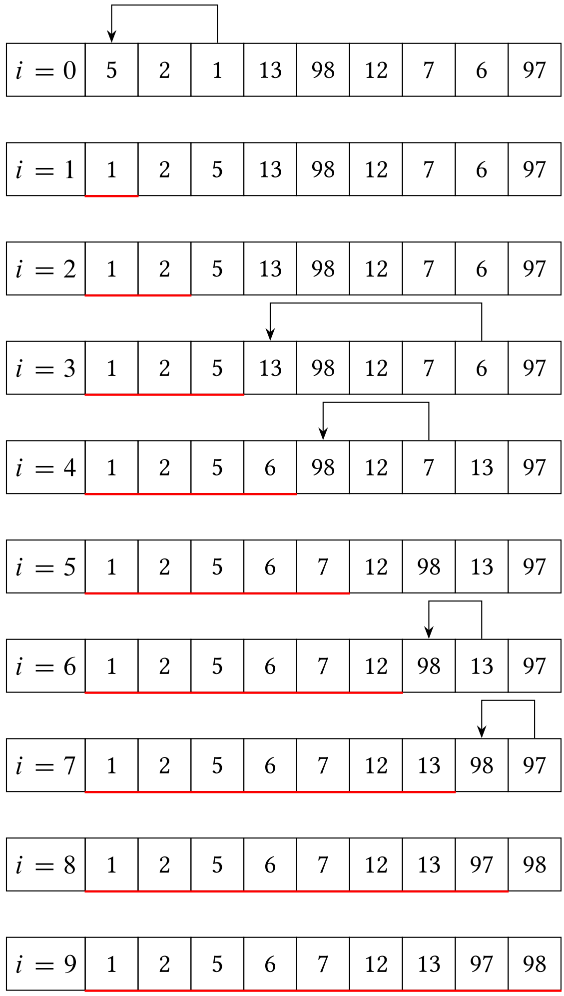


Figure 7.2: In-Place Selection Sort Illustration

```
      // Find the minimum value.  If we get a value less than the current minimum,
      // we need to swap at the end.
      for (int j = i + 1; j < ls.size(); j++) {
        if (ls.get(j).compareTo(min) < 0) {
          min = ls.get(j);
          minIdx = j;
          needToSwap = true;
        }
      }
      // Swap the minimum value with the current index, if need be.
      if(needToSwap) { Collections.swap(ls, minIdx, i); }
    }
    return ls;
  }
}
```

Bubble Sort

The bubble sort is the last of the poor-performing sorts that we will discuss. With bubble sort we repeatedly swap adjacent elements if they are in the wrong order. We repeat this process until the list is sorted, which is guaranteed after at most n^2 iterations, where n is the size of the input list. Figure 7.3 illustrates the in-place bubble sort algorithm. Note the use of two iteration variables i and j, where i represents the outer loop and j represents the inner loop. The inner loop is responsible for the actual swapping of elements, whereas the outer controls the number of traversals over the list. The idea is to "bubble" the elements to the top/end of the list via repeated comparisons and swapping, hence the name. There are optimizations that can be made to the bubble sort algorithm. For example, if we traverse through the entire list without swapping at all, then the list must be in sorted order and we can terminate early. We will not implement this optimization, but it is worth considering.

Listing 7.21—Bubble Sort Tester

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

class BubbleSortTester {

  @Test
  void fbs() {
    IBubbleSort<Integer> fbs = new FunctionalBubbleSort<>();
    assertAll(
      () -> assertEquals(List.of(1, 2, 3, 4, 5), fbs.bubbleSort(LS1)),
      () -> assertEquals(List.of(), fbs.bubbleSort(LS2)),
      () -> assertEquals(List.of(2, 3, 3, 3, 6, 7, 8, 10, 10, 10),
                         fbs.bubbleSort(LS3)));
  }

  @Test
  void ipbs() {
    IBubbleSort<Integer> ipbs = new InPlaceBubbleSort<>();
    assertAll(
      () -> ipbs.bubbleSort(LS1),
      () -> assertEquals(List.of(1, 2, 3, 4, 5), LS1),
      () -> ipbs.bubbleSort(LS2),
      () -> assertEquals(List.of(), LS2),
      () -> ipbs.bubbleSort(LS3),
      () -> assertEquals(List.of(2, 3, 3, 3, 6, 7, 8, 10, 10, 10), LS3));
  }
}
```

Listing 7.22—Bubble Sort Generic Interface

```
import java.lang.Comparable;
import java.util.AbstractList;

interface IBubbleSort<V extends Comparable<V>> {
  AbstractList<V> bubbleSort(AbstractList<V> ls);
}
```

Functional Bubble Sort

The functional bubble sort works similarly to the functional insertion sort. We remove exactly one instance of the largest element in the list, then recursively sort the remaining list. Once we know that list is sorted (by the recursive invariant property), we insert the largest element back into the list.

Listing 7.23—Functional Bubble Sort Implementation

```java
import java.lang.Comparable;
import java.util.AbstractList;
import java.util.ArrayList;

class FunctionalBubbleSort<V extends Comparable<V>> implements IBubbleSort<V> {

  @Override
  public AbstractList<V> bubbleSort(AbstractList<V> ls) {
    if (ls.size() <= 1) { return ls; }
    else {
      // Find the largest element.
      V largest = getLargest(ls);
      boolean removed = false;

      // Get all elements but the largest.  Removes only
      // one occurrence of the largest element.
      AbstractList<V> rest = new ArrayList<>();
      for (V v : ls) {
        if (v.equals(largest) && !removed) { removed = true; }
        else { rest.add(v); }
      }

      // Bubble sort the rest, then add the largest as the last element.
      AbstractList<V> newLs = bubbleSort(rest);
      newLs.add(largest);
      return newLs;
    }
  }

  /**
   * Get the largest element in the list.
   * @param ls the list to search.
   * @return the largest element in the list.
   */
  private V getLargest(AbstractList<V> ls) {
    return ls.stream()
              .max(Comparable::compareTo)
              .orElse(null);
  }
}
```

In-place Bubble Sort

Listing 7.24—In-place Bubble Sort Implementation

```java
import java.lang.Comparable;
import java.util.AbstractList;
import java.util.Collections;

class InPlaceBubbleSort<V extends Comparable<V>> implements IBubbleSort<V> {
```

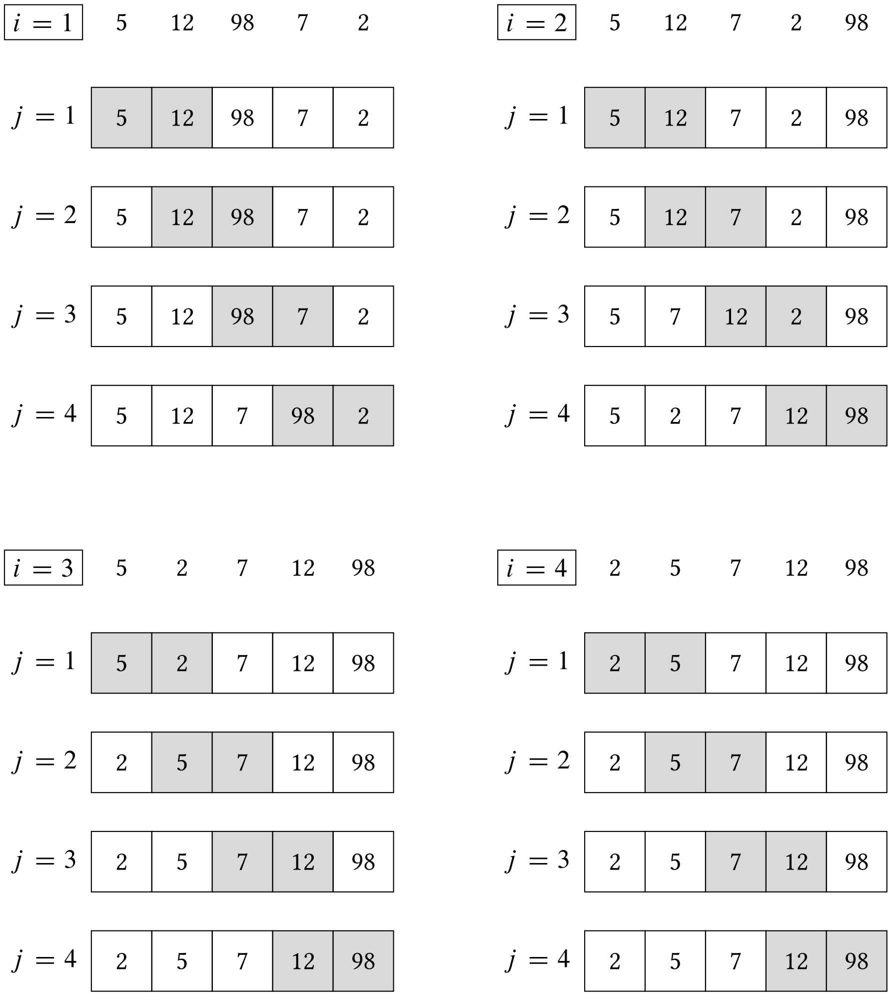


Figure 7.3: In-Place Bubble Sort Illustration

```
  @Override
  public AbstractList<V> bubbleSort(AbstractList<V> ls) {
    for (int i = 0; i < ls.size(); i++) {
      for (int j = 0; j < ls.size() - i - 1; j++) {
        if (ls.get(j).compareTo(ls.get(j + 1)) > 0) {
          Collections.swap(ls, j, j + 1);
        }
      }
    }
    return ls;
  }
}
```

Merge Sort

The *merge sort* is one of the first explicitly divide-and-conquer algorithms that programmers encounter. We divide the sorted list into halves and recursively sort those halves. The base case is when the list is a singleton, since we certainly know how to sort a list that contains less than or equal to one element. After dividing comes the conquering, which consists of taking two now-sorted lists and combining their elements to create yet another sorted list. Recall that, at the base case, we know the singletons are sorted, because they have to be by definition. Because of this *invariant*, we know that merging the contents of two sorted lists is trivial; we compare each element one-by-one, putting the smaller of the two before the subsequent value.

Example 7.2. Consider merging the lists $l_1 = [9, 11, 14]$, and $l_2 = [2, 20]$. We create a new list l_3 whose size is the sum of the sizes of l_1 and l_2. Then we compare 9 against 2, of which the latter is smaller, meaning it goes first in l_3. Then we have 20 against 9, of which the latter is smaller, meaning it is second. Then we have 20 against 11, of which the latter is smaller, meaning it is third. Then we have 20 against 14, of which the latter is, once again, smaller, meaning it is fourth. Since we have exhausted all elements of l_1, and we know for certain that l_2 is sorted, we can then just copy the remaining elements of l_2 into l_3.

Listing 7.25—Merge Sort Tester

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

class MergeSortTester {

  @Test
  void fmsTester() {
    IMergeSort<Integer> ms = new FunctionalMergeSort<>();
    assertAll(
      () -> assertEquals(List.of(1, 2, 3, 4, 5), ms.mergeSort(LS1)),
      () -> assertEquals(List.of(), ms.mergeSort(LS2)),
      () -> assertEquals(List.of(2, 3, 3, 3, 6, 7, 8, 10, 10, 10),
                         ms.mergeSort(LS3)));
  }

  @Test
  void ipmsTester() {
    IMergeSort<Integer> ms = new FunctionalMergeSort<>();
    assertAll(
      () -> ms.mergeSort(LS1),
      () -> assertEquals(List.of(1, 2, 3, 4, 5), LS1),
      () -> ms.mergeSort(LS2),
      () -> assertEquals(List.of(), LS2),
      () -> ms.mergeSort(LS3),
      () -> assertEquals(List.of(2, 3, 3, 3, 6, 7, 8, 10, 10, 10), LS3));
  }
}
```

Listing 7.26—Merge Sort Generic Interface

```java
import java.util.AbstractList;

interface IMergeSort<V extends Comparable<V>> {
  AbstractList<V> mergeSort(AbstractList<V> ls);
```

```
}
```

Functional Merge Sort

Listing 7.27—Functional Merge Sort Implementation

```java
import java.lang.Comparable;
import java.util.AbstractList;
import java.util.ArrayList;

class FunctionalMergeSort<V extends Comparable<V>> implements IMergeSort<V> {

  @Override
  public AbstractList<V> mergeSort(AbstractList<V> ls) {
    return this.msHelper(ls);
  }

  /**
   * Recursive helper method for merge sort. Splits the list in half and merges
   * the two halves after recursively sorting them.
   * @param ls the list to sort.
   * @return the sorted list.
   */
  private AbstractList<V> msHelper(AbstractList<V> ls) {
    if (ls.isEmpty()) { return new ArrayList<>(); }
    else if (ls.size() == 1) {
      AbstractList<V> newLs = new ArrayList<>();
      newLs.add(ls.get(0));
      return newLs;
    } else {
      int mid = (ls.size() - 1) / 2;
      List<V> leftHalf = ls.subList(0, mid + 1);
      List<V> rightHalf = ls.subList(mid + 1, ls.size());
      AbstractList<V> leftHalfSort = this.msHelper((AbstractList<V>) leftHalf);
      AbstractList<V> rightHalfSort = this.msHelper((AbstractList<V>) rightHalf);
      return this.merge(leftHalfSort, rightHalfSort);
    }
  }

  /**
   * Merges two sorted lists into one sorted list. Compares each element
   * one-by-one and adds the smaller element to the new list. If one list
   * is exhausted, the elements of the other list are added to the new list.
   *
   * @param ls1 the first sorted list.
   * @param ls2 the second sorted list.
   * @return the merged sorted list.
   */
  private AbstractList<V> merge(AbstractList<V> ls1, AbstractList<V> ls2) {
    int i, j = 0;
    AbstractList<V> newLs = new ArrayList<>();

    // Merge the lists, comparing the elements.
    while (i < ls1.size() && j < ls2.size()) {
      if (ls1.get(i).compareTo(ls2.get(j)) < 0) { newLs.add(ls1.get(i++)); }
      else { newLs.add(ls2.get(j++)); }
    }

    // Finish copying ls1.
    while (i < ls1.size()) { newLs.add(ls1.get(i++)); }
```

```
    // Finish copying ls2.
    while (j < ls2.size()) { newLs.add(ls2.get(j++)); }

    return newLs;
  }
}
```

In-place Merge Sort

Listing 7.28—In Place Merge Sort

```
import java.lang.Comparable;
import java.util.AbstractList;
import java.util.ArrayList;

class InPlaceMergeSort<V extends Comparable<V>> implements IMergeSort<V> {

  @Override
  public AbstractList<V> mergeSort(AbstractList<V> ls) {
    this.msHelper(ls, 0, ls.size() - 1);
    return ls;
  }

  /**
   * Recursively sorts the list by splitting it in half and merging the two halves
   * @param ls the list to sort.
   * @param low the lower bound of the sublist.
   * @param high the upper bound of the sublist.
   */
  private void msHelper(AbstractList<V> ls, int low, int high) {
    if (low < high) {
      int mid = low + (high - low) / 2;
      this.msHelper(ls, low, mid);
      this.msHelper(ls, mid + 1, high);
      this.merge(ls, low, mid, high);
    }
  }

  /**
   * Merges two sorted sublists into one sorted list.
   * @param ls the list to sort.
   * @param low the lower bound of the sublist.
   * @param mid the middle index of the sublist.
   * @param high the upper bound of the sublist.
   */
  private void merge(AbstractList<V> ls, int low, int mid, int high) {
    AbstractList<V> ls1 = new ArrayList<>();
    AbstractList<V> ls2 = new ArrayList<>();

    for (int i = low; i <= mid; i++) { ls1.add(ls.get(i)); }
    for (int j = mid + 1; j <= high; j++) { ls2.add(ls.get(j)); }

    int mergeIdx = low;
    int i, j = 0;

    // Merge the elements into the existing list.
    while (i < ls1.size() && j < ls2.size()) {
      if (ls1.get(i).compareTo(ls2.get(j)) < 0) {
        ls.set(mergeIdx++, ls1.get(i++));
      } else {
```

```java
        ls.set(mergeIdx++, ls2.get(j++));
      }
    }

    // Copy the rest of the elements over.
    while (i < ls1.size()) { ls.set(mergeIdx++, ls1.get(i++)); }
    while (j < ls2.size()) { ls.set(mergeIdx++, ls2.get(j++)); }
  }
}
```

Quick Sort

At last we arrive at the `quick sort` algorithm. Quick sort works by choosing a *pivot*, which is some value in the list. We then recursively sort all elements that are less than the pivot and all those that are greater than the pivot. Our implementation of the in-place quicksort works slightly differently, which is why we favor the functional version over the in-place counterpart.

Quick sort performs optimally when the median is selected as the pivot because roughly half of the elements are less than the pivot and roughly half are greater than the pivot, allowing for a performance similar to that of merge sort. Unfortunately, finding the median a priori to sorting ultimately defeats the point. In the subsequent chapter we will analyze the sorting algorithms in more detail.

Listing 7.29—Quick Sort Tester

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

class QuickSortTester {

  @Test
  void fqsTester() {
    IQuickSort<Integer> fqs = new FunctionalQuickSort<>();
    assertAll(
      () -> assertEquals(List.of(1, 2, 3, 4, 5), fqs.quicksort(LS1)),
      () -> assertEquals(List.of(), fqs.quicksort(LS2)),
      () -> assertEquals(List.of(2, 3, 3, 3, 6, 7, 8, 10, 10, 10),
                         fqs.quicksort(LS3)));
  }

  @Test
  void ipqsTester() {
    IQuickSort<Integer> ipqs = new InPlaceQuickSort<>();
    assertAll(
      () -> ipqs.quicksort(LS1),
      () -> assertEquals(List.of(1, 2, 3, 4, 5), LS1),
      () -> ipqs.quicksort(LS2),
      () -> assertEquals(List.of(), LS2),
      () -> ipqs.quicksort(LS3),
      () -> assertEquals(List.of(2, 3, 3, 3, 6, 7, 8, 10, 10, 10), LS3));
  }
}
```

Listing 7.30—Quick Sort Generic Interface

```java
import java.lang.Comparable;
import java.util.AbstractList;
```

```java
interface IQuickSort<V extends Comparable<V>> {
  AbstractList<V> quicksort(AbstractList<V> ls);
}
```

Functional Quick Sort

The functional implementation of quick sort is beautiful and elegant. We choose a pivot p at random, then create three sublists $l_<$, $l_>$, $l_=$, where $l_<$ stores all elements less than p, where $l_>$ stores all elements greater than p, and $l_=$ stores all elements equal to the pivot. Each sublist, excluding $l_=$, is recursively sorted, followed by concatenating the three sublists in order.

Listing 7.31—Functional Quick Sort Implementation

```java
import java.lang.Comparable;
import java.util.AbstractList;
import java.util.stream.Collectors;

class FunctionalQuickSort<V extends Comparable<V>> implements IQuickSort<V> {

  @Override
  public AbstractList<V> quicksort(AbstractList<V> ls) {
    if (ls.isEmpty()) { return ls; }
    else {
      // Choose a random pivot.
      V pivot = ls.get((int) (Math.random() * ls.size()));

      // Sort the left-half.
      AbstractList<V> leftHalf = (AbstractList<V>)
                                ls.stream()
                                  .filter(x -> x.compareTo(pivot) < 0)
                                  .collect(Collectors.toList());
      AbstractList<V> leftSorted = quicksort(leftHalf);

      // Sort the right-half.
      AbstractList<V> rightHalf = (AbstractList<V>)
                                 ls.stream()
                                   .filter(x -> x.compareTo(pivot) > 0)
                                   .collect(Collectors.toList());
      AbstractList<V> rightSorted = quicksort(rightHalf);

      // Get all elements equal to the pivot.
      AbstractList<V> equal = (AbstractList<V>)
                             ls.stream()
                               .filter(x -> x.compareTo(pivot) > 0)
                               .collect(Collectors.toList());

      // Merge the three.
      leftSorted.addAll(equal);
      leftSorted.addAll(rightSorted);
      return leftSorted;
    }
  }
}
```

In-place Quick Sort

Listing 7.32—In-place Quick Sort Implementation

```java
import java.lang.Comparable;
import java.util.*;

class InPlaceQuickSort<V extends Comparable<V>> implements IQuickSort<V> {

  @Override
  public AbstractList<V> quicksort(AbstractList<V> ls) {
    this.quicksortHelper(ls, 0, ls.size() - 1);
    return ls;
  }

  /**
   * Recursive helper method for quicksort.
   * @param ls the List to sort.
   * @param low the lower bound of the partition.
   * @param high the upper bound of the partition.
   */
  private void quicksortHelper(AbstractList<V> ls, int low, int high) {
    if (low < high) {
      int pivot = quicksortPartition(ls, low, high);
      quicksortHelper(ls, low, pivot - 1);
      quicksortHelper(ls, pivot + 1, high);
    }
  }

  /**
   * Creates a quicksort partition, where all elements less than the pivot are
   * to the left of the pivot, and all elements greater than the pivot are to its
       right.
   * @param ls the List to partition.
   * @param low the lower bound of the partition.
   * @param high the upper bound of the partition.
   * @return the index of the pivot.
   */
  private int quicksortPartition(AbstractList<V> ls, int low, int high) {
    int rand = new Random().nextInt(high - low + 1) + low;
    Collections.swap(ls, rand, high);
    V pivot = ls.get(high);
    int prevLowest = low;
    for (int i = low; i <= high; i++) {
      if (ls.get(i).compareTo(pivot) < 0) {
        Collections.swap(ls, i, prevLowest++);
      }
    }
    Collections.swap(ls, prevLowest, high);
    return prevLowest;
  }
}
```

8. Algorithm Analysis

8.1 Analyzing Algorithms

Asymptotic analysis, or in general, the analysis of function growth, relates heavily to the performance of an algorithm. We can represent algorithms as mathematical functions, and describe their relative performance in terms of the input size.

Example 8.1. Consider the linear search algorithm. We know that, in the best case, the item that we are searching for is the first element in the list. In the average case, it is found in the middle of the list. In the worst case, the element does not exist in the list. Because we have to traverse through n elements, namely the n elements of our input list, we say that the linear search grows in linear proportion to its input size. *Best-case*, *average-case*, and *worst-case* describe the potential inputs to a function. We can ascribe the same meanings to binary search, the sorting algorithms from the previous chapter, and beyond. Though, we need a notation to denote the growth rate of a function. In computer science we most often make use of Big-Oh notation, which denotes a function's upper bound. That is, a function $f(n) = \mathcal{O}(g(n))$ if, at some point, $f(n)$ begins to forever grow slower than or equal to $g(n)$. We can roughly replace the equals '$=$' sign with a less-than-or-equal-to '$\leq$' sign. One detail to note is that the end-behavior of a polynomial is determined by its highest-order term. For example, $f(n) = 0.5x^2 + 0.5\cos(\mathsf{deg}(5x))$ is upper-bounded by $g(n) = 0.5x^2$, because the cosine function is upper-bounded by 1. Thus, when describing a function in asymptotic terms, we can drop/ignore all coefficients and lower-order terms.

Example 8.2. Consider the following functions $f(n) = 0.5x^2 + 0.5\cos(\mathsf{deg}(5x))$ and $g(n) = 0.2x^3 + 0.3\sin(\mathsf{deg}(4x))$. As we stated, we can drop all lower-order terms and coefficients, meaning $f(n) = n^2$ and $g(n) = n^3$. From here, it is trivial to see that for any $n > 1$, $g(n)$ grows faster than $f(n)$ Thus, we say that $f(n) = \mathcal{O}(g(n))$.[1]

[1] There is a bit more to the formalism of Big-Oh, but we will explain those details in due time.

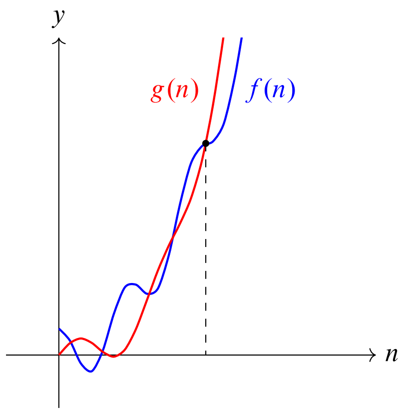


Figure 8.1: $f(n) = \mathcal{O}(g(n))$

There are two other common notations for asymptotic analysis: Big-Omega and Theta. Big-Omega describes the lower-bound of a function, and Theta describes the tight bound of a function. That is, $f(n) = \Omega(g(n))$ if there is a point at which $f(n)$ begins to grow faster than or equal to $g(n)$. We can roughly replace the equals '$=$' sign with a greater-than-or-equal-to '$\geq$' sign. Similarly, $f(n) = \Theta(g(n))$ if $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$. We can roughly replace the equals '$=$' sign with an equivalence '$\equiv$' sign. Examples for Omega and Theta are harder to come by without a formalized definition, so we will defer them until later.

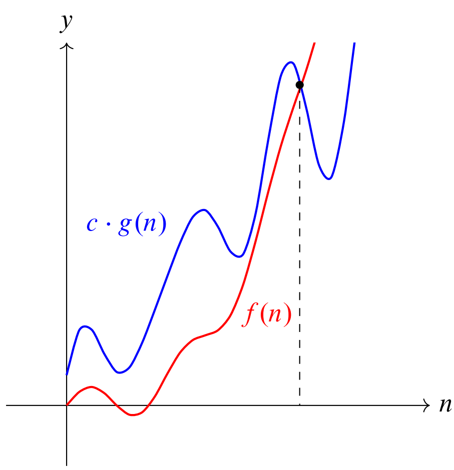


Figure 8.2: $f(n) = \Omega(g(n))$

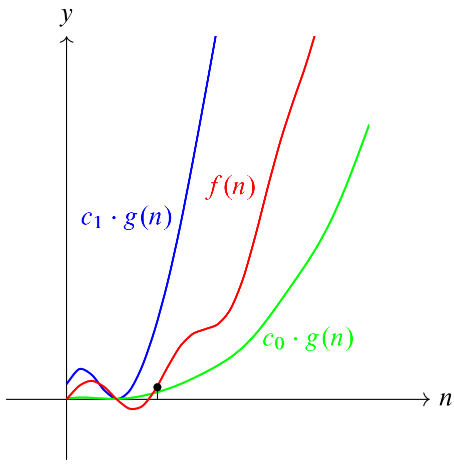


Figure 8.3: $f(n) = \Theta(g(n))$

The term “asymptotic analysis” stems from the fact that “asymptotic” behavior describes the behavior of a function as its input approaches infinity.

Programmers often conflate Big-Oh as meaning the “worst-case” of an algorithm, Big-Omega as the “best case,” and Theta as the “average case.” In actuality, these have no precise and deterministic relation to one another. Remember that Big-Oh is the upper-bound, Big-Omega is the lower-bound, and Theta exists if and only if the function is Big-Oh and Big-Omega of the same function.

Example 8.3. Consider the following function.

```
// foo receives an array of integers.
foo(ls) {
  // v is a random integer between 0 and 100, with equal probability.
  v := RNG()
  n := len(ls)
  if v == 100
    return 42
  else if v == 0
    for i := 0 to n do
      for j := 0 to n do
        n := n + i * j;
  else
    for i := 0 to n do
      n := n + i;
  return n;
}
```

The best-case for this algorithm is for v to be 100, meaning we immediately return 42. Therefore, we are upper-bounded by $\mathcal{O}(1)$, since this is a constant-time operation. We are similarly lower-bounded by this operation, meaning it is $\Omega(1)$. Therefore it is also $\Theta(1)$ in the best case.

In the worst case, v is zero, meaning we have a nested for-loop over the length of the input list, meaning we are (strictly) upper-bounded by $\mathcal{O}(n^2)$. The `for`-loops must always execute, meaning that we are lower-bounded by $\Omega(n^2)$ as well. We can conclude similar reasoning for $\Theta(n^2)$.

In the average case, i.e., when n is neither 0 nor 100, we loop once over the length of the input list, meaning we are both upper and lower-bounded by $\mathcal{O}(n)$ and $\Omega(n)$ respectively.

Example 8.4. Recall the binary search algorithm. In the best case, we find the element immediately. Therefore we can conclude that we are lower-bounded by $\Omega(1)$, but we can also reasonably conclude that we are upper-bounded by $\mathcal{O}(n^3)$. This seems odd, but it's certainly true; finding the element immediately will never exceed $\mathcal{O}(n^3)$. It's fair to conclude that we are upper-bounded by $\mathcal{O}(n^n)$ as well; we will never grow faster than $\mathcal{O}(n^n)$. These are what we call non-strict upper-bounds, and are a bit sloppy to state for binary search. In the best case, we find the element immediately, so saying anything other than that the upper-bound is $\mathcal{O}(1)$ is, while technically correct, loose. Moreover, in these instances, should we say that the upper-bound is not $\mathcal{O}(1)$ in the best case, we lose the ability to conclude that the algorithm, in the best case, is $\Theta(1)$.

In the worst case, the element is not in the list. Therefore we are upper-bounded by $\mathcal{O}(\lg n)$, but lower-bounded, yet again, by $\Omega(\lg\lg n)$. This is similarly a sloppy argument to make, because while it is true that we will never find the element faster that $\Omega(\lg\lg n)$ in the worst case, it's not a tight lower bound, meaning we lose the ability to use Theta notation, should we opt for this lower bound.

In the average case, we land somewhere in the middle of finding the element immediately and it not existing at all, meaning that we are upper-bounded by $\mathcal{O}(\lg n)$, and lower-bounded bounded by $\Omega(1)$. Therefore concluding $\Theta(\lg n)$ is incorrect for binary search.

Example 8.5. Recall the insertion sort. We can analyze that, in the best case, the structure is already sorted, meaning nothing needs to be recursively sorted and inserted. Therefore, we require exactly one traversal over the data, meaning it is upper-bounded by $\mathcal{O}(n)$. Additionally, because we do require exactly one traversal over the input data, we are also lower-bounded by $\Omega(n)$. Hence, we can also conclude that, in the best-case, insertion sort is $\Theta(n)$.

In the worst-case, the list is in reversed order. So, we must insert each element in the correct position, with respect to every other element. Therefore we are upper and lower-bounded by $\mathcal{O}(n^2)$ and $\Omega(n^2)$ respectively.

Formalizing Big-Oh, Big-Omega, and Theta

We have described the Big-Oh, Big-Omega, and Theta notations informally. When proving the asymptotic bounds of a mathematical function, we often need a formal proof thereof. We will now describe the formalism of these notations.

Big-Oh

A function $f(n) = \mathcal{O}(g(n))$ if there exists a constant $c > 0$ and a point n_0 such that $f(n) \leq cg(n)$ for every n greater than or equal to n_0. Interestingly, we can describe all three notations in terms of limits. We can say that $f(n) = \mathcal{O}(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty$. Unfortunately there is no hard-and-fast rule to apply when finding the c and n_0 constants. What is convenient about it, however, is that multiple solutions may work, hence the existential quantifiers in the equality definition.

$\{\exists c > 0\}(\exists n_0(\forall n > n_0 \to f(n) \leq cg(n)))$

Example 8.6. Prove that $3n^2 + 6n = \mathcal{O}(n^2)$. We need to find a constant $c > 0$ and a point n_0 such that $3n^2 + 6n \leq cn^2$ for every $n \geq n_0$. Let's move $3n^2$ to the right-hand side of the inequality, and divide both sides by n^2.

$$\begin{aligned}
3n^2 + 6n &\leq cn^2 \\
6n &\leq cn^2 - 3n^2 \\
6n &\leq n^2(c-3) \\
\frac{6n}{n^2} &\leq c - 3 \\
\frac{6}{n} &\leq c - 3 \\
\frac{6}{n} + 3 &\leq c \\
c &\geq \frac{6}{n} + 3
\end{aligned}$$

If we assign n to be 1, then $c \geq 9$, and the inequality holds for all $n \geq 1$. Therefore we can conclude that $3n^2 + 6n = \mathcal{O}(n^2)$. We can also evaluate this as a limit:

$$\begin{aligned}
\lim_{n\to\infty} \frac{3n^2 + 6n}{n^2} &= \lim_{n\to\infty} \frac{3n^2}{n^2} + \frac{6n}{n^2} \\
&= \lim_{n\to\infty} 3 + \frac{6}{n} \\
&= 3 + 0 \\
&= 3 < \infty
\end{aligned}$$

Example 8.7. Prove that $0.25n^4 - 6000n^3 + 25 \neq \mathcal{O}(n^3)$. To show that this relationship does not hold, we can do either a proof-by-contradiction or use the limit definition. Let's do a proof-by-contradiction. Assume the contrary, that $0.25n^4 - 6000n^3 + 25 = \mathcal{O}(n^3)$. Then there exists a constant $c > 0$ and a point n_0 such that $0.25n^4 - 6000n^3 + 25 \leq cn^3$ for every $n \geq n_0$.

$$\begin{aligned}
0.25n^4 - 6000n^3 + 25 &\leq cn^3 \\
25 &\leq cn^3 - 0.25n^4 + 6000n^3 \\
25 &\leq n^3(c + 0.25n + 6000) \\
\frac{25}{n^3} &\leq c + 0.25n + 6000 \\
c &\geq \frac{25}{n^3} - 0.25n - 6000
\end{aligned}$$

The problem here is that, no matter what constant we choose for c, there is always an n that will falsify the inequality. Therefore we can conclude that $0.25n^4 - 6000n^3 + 25 \neq \mathcal{O}(n^3)$. We can also evaluate this as a limit:

$$\begin{aligned}
\lim_{n\to\infty} \frac{0.25n^4 - 6000n^3 + 25}{n^3} &= \lim_{n\to\infty} \frac{0.25n^4}{n^3} - \frac{6000n^3}{n^3} + \frac{25}{n^3} \\
&= \lim_{n\to\infty} 0.25n - 6000 + \frac{25}{n^3} \\
&= \infty - 6000 + 0 \\
&= \infty
\end{aligned}$$

Example 8.8. Prove that $(2n^2 + n)(4n) = \mathcal{O}(n^3)$. Expanding this expression, we get $8n^3 + 4n^2$, meaning we need to find constants c and n_0 such that $8n^3 + 4n^2 \leq cn^3$ for all $n \geq n_0$.

$$\begin{aligned}
8n^3 + 4n^2 &\leq cn^3 \\
4n^2 &\leq cn^3 - 8n^3 \\
4n^2 &\leq n^3(c - 8) \\
\frac{4n^2}{n^3} &\leq c - 8 \\
\frac{4}{n} &\leq c - 8 \\
\frac{4}{n} + 8 &\leq c \\
c &\geq \frac{4}{n} + 8
\end{aligned}$$

For $n_0 = 1$, we have $c \geq 12$. Therefore we can conclude that $(2n^2 + n)(4n) = \mathcal{O}(n^3)$. We can also evaluate this as a limit:

$$\begin{aligned}
\lim_{n\to\infty} \frac{(2n^2 + n)(4n)}{n^3} &= \lim_{n\to\infty} \frac{8n^3 + 4n^2}{n^3} \\
&= \lim_{n\to\infty} 8 + \frac{4}{n} \\
&= 8 + 0 \\
&= 8 < \infty
\end{aligned}$$

Example 8.9. Prove that $(4n)^n \neq \mathcal{O}(n^n)$. Assume to the contrary that $(4n)^n = \mathcal{O}(n^n)$. Then there exists a constant $c > 0$ and a point n_0 such that $(4n)^n \leq cn^n$ for every $n \geq n_0$. Expanding the left-hand side, then dividing both sides by n^n gets us:

$$\begin{aligned}
4^n \cdot n^n &\leq cn^n \\
4^n &\leq c
\end{aligned}$$

The problem is that we cannot pick a constant c without finding an n that falsifies the inequality. Therefore, by contradiction, $(4n)^n \neq \mathcal{O}(n^n)$.

Example 8.10. Prove that $3n + n \lg n = \mathcal{O}(n^2)$. Dividing both sides of the equation by n gets us:

$$\begin{aligned} 3n + n \lg n &\leq cn^2 \\ 3 + \lg n &\leq cn \\ \lg n &\leq cn - 3 \end{aligned}$$

Suppose $c = 3$. Then, $\lg n \leq n$ for any n greater than 2, so we can set $n_0 = 3$. Thus, $3n + n \lg n = \mathcal{O}(n^2)$.

Big-Omega

A function $f(n) = \Omega(g(n))$ if there exists a constant $c > 0$ and a point n_0 such that $f(n) \geq cg(n)$ for every n greater than or equal to n_0. We can describe this in terms of limits as well. We can say that $f(n) = \Omega(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0$.

$$\boxed{\{\exists c > 0\}(\exists n_0(\forall n \geq n_0 \to f(n) \geq cg(n)))}$$

Example 8.11. Prove that $(2n^3 - 6n^2) = \Omega(n^2)$. We need to find a constant $c > 0$ and a point n_0 such that $(2n^3 - 6n^2) \geq cn^2$ for every $n \geq n_0$. First, let's factor out the n^2 on the left-hand side of the inequality.

$$\begin{aligned} 2n^3 - 6n^2 &\geq cn^2 \\ n^2(2n - 6) &\geq cn^2 \\ 2n - 6 &\geq c \\ c &\leq 2n - 6 \end{aligned}$$

We know that $2n - 6$ is always greater than 0 for $n > 3$, so we will pick $n_0 = 4$ and $c_0 = 1$. Therefore we can conclude that $(2n^3 - 6n^2) = \Omega(n^2)$.

Example 8.12. Prove that $3n^2 + 4n - 8 = \Omega(n^2)$. We need to find a constant $c > 0$ and a point n_0 such that $3n^2 + 4n - 8 \geq cn^2$ for every $n \geq n_0$. We can divide both sides by n^2.

$$\begin{aligned} 3n^2 + 4n - 8 &\geq cn^2 \\ 3 + \frac{4}{n} - \frac{8}{n^2} &\geq c \\ c &\leq 3 + \frac{4}{n} - \frac{8}{n^2} \end{aligned}$$

Choosing $n_0 = 8$, we need a value of c to satisfy the inequality $c \leq 3 + \frac{1}{2} - \frac{1}{64}$, which reduces to $c \leq 3.484375$. So, picking $c = 3$ works, and we have proved that $3n^2+4n-8 = \Omega(n^2)$.

Theta

A function $f(n) = \Theta(g(n))$ if there exists constants $c_0, c_1 > 0$ and a point n_0 such that $c_0 g(n) \leq f(n) \leq c_1 g(n)$ for every n greater than n_0. We can also say that $f(n) = \Theta(g(n))$ if $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$.

$$\boxed{\exists\{c_0, c_1 > 0\}(\exists n_0(\forall n > n_0 \to c_0 g(n) \leq f(n) \leq c_1 g(n)))}$$

Misconceptions About Asymptotic Analyses

As we mentioned before, many programmers conflate best, average, and worst-cases with Big-Omega, Theta, and Big-Oh respectively. There is no discernible relationship between these concepts.

Example 8.13. Consider the absolutely egregious statement, "linear search is n." The big problem here is that we are using n without any qualification; n what? A slightly better, but still poor, way to phrase this is, "linear search is $\mathcal{O}(n)$," which adds the upper-bound. The problem now is that we have yet to state under what conditions is linear search $\mathcal{O}(n)$, i.e., best-case, average-case, worst-case. So, we can state, "linear search, in the worst-case, is $\mathcal{O}(n)$." Even though this is an accurate statement, using a loose upper-bound when the lower-bound is known and is equal to the upper-bound is sloppy. We can say, "linear search, in the worst-case, is $\Theta(n)$," which is a tight bound. In summary, being specific about the conditions under which an algorithm is $\mathcal{O}(n)$, $\Omega(n)$, or $\Theta(n)$ is important, as is using tightened bounds when possible.

Example 8.14. Consider the statement, "Insertion sort is faster than merge sort since it is $\mathcal{O}(n)$ while merge sort is $\Theta(n \lg n)$." There are two problems with such a claim: first, it omits the qualification of what case analysis we wish to reference. To fix this, we should add "in the best case" immediately after "is faster than merge sort." Second, we could tighten the bound of insertion sort because it is $\mathcal{O}(n)$ and $\Omega(n)$ in the best case.

Example 8.15. Consider the statement, "The worst-case running time of selection sort is $\mathcal{O}(n^2)$ and the worst-case running time of merge sort is $\mathcal{O}(n \lg n)$; therefore, merge sort is asymptotically faster in the worst-case." Is this statement correct? Unfortuntately, it is not, and we can fix it by changing only the asymptotic functions. It is incorrect because Big-Oh only describes the upper-bound of a function. We cannot conclude that merge sort is asymptotically faster in the worst-case because we do not know the lower-bound of either algorithm. To correct the statement, we can ascribe a tight-bound on the growth of the functions via $\Theta(n^2)$ and $\Theta(n \lg n)$ respectively. We could also just place the tight-bound on only $\Theta(n^2)$, which then provides the lower-bound of selection sort, but as we stated before, using tight-bounds is the preferred option.

Analysis of the Sorting Algorithms

We can analyze the five sorting algorithms described in Chapter 7 in terms of their asymptotic behavior in the best, average, and worst cases.

Bubble Sort

Starting off with bubble sort, in the best case, the array is already sorted, meaning we only need to traverse the array once. Therefore, we are upper and lower-bounded by $\mathcal{O}(n)$ and $\Omega(n)$ respectively. Hence, we can conclude that bubble sort is $\Theta(n)$ in the best case.

In the average case, each element is roughly "half way" to its sorted position. We can compute the expected number of swaps/inversions as follows: an array of length n has an inversion $I_{i,j} = 1$ if we must swap the values (i, j). Therefore the expected value of there being an inversion for any arbitrary pair is $1/2$ because either a pair must or must not be inverted. Our loop traverses from $i = 1$ to n, with an inner loop of $j > i$ to In the average case, each element is roughly "half way" to its sorted position. We can compute the expected number of swaps/inversions as follows: an array of length n has an inversion $I_{i,j} = 1$ if we must swap the values (i, j). Therefore the expected value of there being an inversion for any arbitrary pair is $1/2$ because either a pair must or must not be inverted. Our loop traverses from $i = 1$ to n, with an inner loop of $j > i$ to n, both of which correspond to summations. Because the inner term depends on neither i nor j, we need to determine how many pairs are such that $1 \leq i < j \leq n$. We are, in effect, choosing two values out of a possible n, which collapses to $\binom{n}{2}$, which resolves to $\dfrac{n(n-1)}{2}$.

$$
\begin{aligned}
\mathbf{E}(\sum_{i=1}^{n}\sum_{j>i}^{n} I_{i,j}) &= \sum_{i=1}^{n}\sum_{j>i}^{n}\frac{1}{2} \\
&= \binom{n}{2}\cdot\frac{1}{2} \\
&= \frac{n(n-1)}{2}\cdot\frac{1}{2} \\
&= \frac{n(n-1)}{4} \\
&= \frac{n^2}{4} - \frac{n}{4}
\end{aligned}
$$

Dropping lower-order terms and coefficients shows that, in the average case, we are upper and lower-bounded by $\mathcal{O}(n^2)$ and $\Omega(n^2)$ respectively. Hence, we can conclude that bubble sort is $\Theta(n^2)$ in the average case.

In the worst case, the array is sorted in reverse order, meaning we must traverse the array n times, and each traversal requires n swaps. Therefore, we are upper and lower-bounded by $\mathcal{O}(n^2)$ and $\Omega(n^2)$ respectively. Hence, we can conclude that bubble sort is $\Theta(n^2)$ in the worst case.

Selection Sort

Up next we come to selection sort, which as we know from the previous chapter, always finds the minimum element and places it at the beginning of the array. Finding the minimum element requires n comparisons, and we must do this for every element in the list. Therefore, in all cases, no matter the input, we are upper and lower-bounded by $\mathcal{O}(n^2)$ and $\Omega(n^2)$ respectively. Hence, we can conclude that selection sort is $\Theta(n^2)$ in the best, average, and worst cases. Moreover, we now understand why selection sort is considerably worse than the other four sorting algorithms, because even in the best case, its asymptotic time complexity is still $\Theta(n^2)$.

Insertion Sort

Insertion sort, similar to bubble sort, has a good start with its best case. The in-place insertion sort algorithm traverses through the list, and makes swaps with out-of-order elements. Therefore, in the best case, the list is already sorted, meaning it traverses over the data exactly once, meaning it is upper and lower-bounded by $\mathcal{O}(n)$ and $\Omega(n)$ respectively. Hence, we can conclude that insertion sort is $\Theta(n)$ in the best case.

In the average case, we perform a similar analysis to bubble sort, in which we determine that every element is "roughly halfway" sorted. This brings about the conclusion that we are upper and lower-bounded by $\mathcal{O}(n^2)$ and $\Omega(n^2)$ respectively. Hence, we can conclude that insertion sort is $\Theta(n^2)$ in the average case.

In the worst case, the list is sorted in reverse order, and every element must be swapped down to the i^{th} index, starting from 1 up to n. So, the element at the last index is swapped n times, the element at the second-to-last index is swapped $n-1$ times, and so on.

$$\begin{aligned}\sum_{i=1}^{n} i &= 1 + 2 + \cdots + (n-1) + n \\ &= \frac{n(n+1)}{2} \\ &= \frac{n^2}{2} + \frac{n}{2}\end{aligned}$$

Therefore, we are upper and lower-bounded by $\mathcal{O}(n^2)$ and $\Omega(n^2)$ respectively. Hence, we can conclude that insertion sort is $\Theta(n^2)$ in the worst case.n, both of which correspond to summations.

Merge Sort

Merge sort is a bit more complicated to analyze, but we can do so by using a recurrence relation. We know that merge sort splits the input list into two halves, and recursively sorts each half. We also know that the base case is when the input list is of length 1, in which case we return the list. Accordingly, we can write the recurrence relation, as a function $T(n)$, as follows:

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \leq 1 \\ 2T(n/2) + \Theta(n), & \text{if } n > 1 \end{cases}$$

Using this definition, we can continuously expand the recurrence relation to determine its asymptotic behavior.

$$\begin{aligned}T(n) &= 2T(n/2) + \Theta(n) \\ &= 2(2T(n/4) + \Theta(n)) + \Theta(n) \\ &= 2(2(2T(n/8) + \Theta(n)) + \Theta(n)) + \Theta(n) \\ &= 2^k T(n/2^k) + k \cdot \Theta(n)\end{aligned}$$

At this point, we have a relationship that is dependent on k, representing the depth of the recurrence. Suppose $n = 2^k$. This implies that $\lg n = k$, because of the base two logarithm properties. Therefore, after substituting we get

$$\begin{aligned}
T(n) &= 2^k T(2^k/2^k) + k \cdot \Theta(n) \\
&= 2^k T(1) + k \cdot \Theta(n) \\
&= \lg(n) \cdot \Theta(1) + \lg(n) \cdot \Theta(n) \\
&= \Theta(\lg(n)) + \Theta(n \lg n) \\
&= \Theta(n \lg n)
\end{aligned}$$

So, we can conclude that merge sort is $\Theta(n \lg n)$ in the best, average, and worst cases. We make this conclusion because, no matter the input, we always subdivide the input into two halves, and merge the two halves together.

Quick Sort

Finally, we will analyze the quick sort algorithm. In the best case, the pivot is always the median, indicating that half of the data is to either of its sides. This relationship resolves to the recurrence relation of the merge sort, whose analysis was in the previous section. Therefore, in the best case, the quick sort time complexity is $\Theta(n \lg n)$.

Jumping down to the worst case, the pivot is always either the minimum or the maximum, meaning that all of the data is to one side of the pivot. As a piecewise equation, we know that the base case of quicksort is $T(n) = 1$ if $n \leq 1$. So, we get

$$T(n) = \begin{cases} \Theta(1), & \text{if } n \leq 1 \\ T(n-1) + \Theta(n), & \text{if } n > 1 \end{cases}$$

The added $\Theta(n)$ accounts for the time partitioning the list, which is linear in terms of the input size. Solving the recurrence relation gets us

$$\begin{aligned}
T(n) &= T(n-1) + \Theta(n) \\
&= (T(n-2) + \Theta(n-1)) + \Theta(n) \\
&= ((T(n-3) + \Theta(n-2)) + \Theta(n-1)) + \Theta(n) \\
&\vdots \\
&= T(1) + \Theta(2) + \Theta(3) + \ldots + \Theta(n-1) + \Theta(n) \\
&= 1 + 2 + 3 + \ldots + (n-1) + n
\end{aligned}$$

The result is an arithmetic series $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$, which resolves to $\Theta(n^2)$ after dropping constants and lower-order terms. Therefore, in the worst case, the quicksort time complexity is $\Theta(n^2)$.

The average case is significantly harder to analyze and the full proof goes beyond the scope of this book. It is known that, in the average case, quicksort is $\Theta(n \lg n)$.

Lower Bound for Comparison-Based Sorting Algorithms

The performance and time complexity of sorting algorithms largely depend on the underlying implementation. We will now prove that, for any comparison-based sorting algorithm, i.e., a sorting algorithm that answers "YES" or "NO" to the question, "Is $a_i < a_j$?" for any list a and indices i and j, it must perform $\Omega(n \lg n)$ comparisons to sort n elements.

Proof. We need two auxiliary lemmas, or true statements, to prove our theorem.

(a) There are $n!$ ways to permute a list of distinct elements $\langle x_1, x_2, \ldots, x_n \rangle$.

(b) Exactly one of these permutations is the correct ordering such that each element $x_{i+1} > x_i$ for all i in $0 \leq i \leq n-1$.

Assume that we have a set S containing answers to every question (i.e., the "YES"/"NO" question we described above) found so far when attempting to sort using comparisons. Of course, by definition, this set must be such that $|S| = n!$ to start. If we answer the first question, we create two groups S_1 and S_2 describing those inputs for which the answer is "YES" and those inputs for which the answer is "NO." Each time we make a decision, we half the problem size, meaning it becomes a logarithmic relationship. In the end, we reach the base case of $|S| = 1$, which means the algorithm knows which output is correct. Thus,

$$\lg n + \lg (n-1) + \ldots + \lg 2 = \lg n! \tag{8.1}$$

$$= \Omega(n \lg n) \tag{8.2}$$

Recall the definition of logarithms: $\lg a + \lg b = \lg ab$. Thus, $\lg a + \lg b + \lg c = \lg abc$, and so forth ad infinitum. Therefore we get the equivalence shown in line (8.1). Using Stirling's approximation, we get the equivalence in line (8.2). QED.

9. Modern Java & Advanced Java Topics

9.1 Verbosity

All the way in the first chapter, we introduced the `main` method, and stated that all methods must belong inside a class. Prior to Java version 21, this was indeed true. Java 21, however, introduced a much cleaner syntax for writing the main method that does not need `public`, nor `static`, nor the input array of strings. These changes make Java much more beginner-friendly. Let's see what a "Hello, world!" program looks like with the improvements.

Listing 9.1—Less Verbose Main Method

```java
class MainMethod {

  void main() {
    System.out.println("Hello, world!");
  }
}
```

Additionally, this change means that any methods that we want to reference/invoke within `main` (that are defined inside the respective class) do not need to be categorized as static. For instance, let's once again write the `double fahrenheitToCelsius(double f)` method, only this time, we will call it from inside the `main` method.

Listing 9.2—???

```java
class MainMethod {

  /**
   * Converts a temperature in Fahrenheit to Celsius.
   * @param f - degrees Fahrenheit.
   * @return degrees Celsius.
   */
  double fahrenheitToCelsius(double f) {
    return (f - 32) * (5.0 / 9.0);
  }

  void main() {
```

```
    System.out.printf("%d deg F = %d deg C.\n", 32, fahrenheitToCelsius(32));
  }
}
```

Whether or not we label `fahrenheitToCelsius` as static is irrelevant in this circumstance. Unfortunately, if we want to write unit tests for this method, it needs to be `static`, as we cannot reference it outside the class definition without the static qualifier. Though, what if we could write unit tests within the `MainMethod` class, rather than writing an entirely separate file? Making this alteration couples the logic of the method with the tests, which, in a large project is not an advisable choice.

To write the unit test, all we need is the `void testFahrenheitToCelsius()` method and prepend the `@Test` annotation.[1] Then, our IDE will automatically detect that we have a test method in the file and is executable.[2] Since we wish to emphasize writing tests before the (method) implementation, we will place the testing method above the method that it tests.

Listing 9.3—Adding Unit Tests in the Class Definition

```java
import static Assertions.assertAll;
import static Assertions.assertEquals;
class MainMethod {

  @Test
  void testFahrenheitToCelsius() {
    assertAll(
      () -> assertEquals(0, fahrenheitToCelsius(32));
      () -> assertEquals(-40, fahrenheitToCelsius(-40));
      () -> assertEquals(100, fahrenheitToCelsius(212));
    )
  }

  /**
   * Converts a temperature in Fahrenheit to Celsius.
   * @param f - degrees Fahrenheit.
   * @return degrees Celsius.
   */
  double fahrenheitToCelsius(double f) {
    return (f - 32) * (5.0 / 9.0);
  }
}
```

In addition to a less-verbose `main` method, Java 21 also introduces *nameless/anonymous classes*, which reduces the required keystrokes even further by removing the need for a class definition. So, if all we want to do is write the `main` method (and perhaps other methods callable from `main`), we might write the following:

Listing 9.4—Adding Unit Tests in the Class Definition

```java
/**
 * Converts a temperature in Fahrenheit to Celsius.
 * @param f - degrees Fahrenheit.
 * @return degrees Celsius.
 */
```

[1] The respective import statements are also necessary.

[2] We will note that we could have done this back in Chapter 1, but we favor separating the tests and the method definition, even if this means we must use the `static` keyword.

```java
double fahrenheitToCelsius(double f) {
  return (f - 32) * (5.0 / 9.0);
}

void main() {
  System.out.printf("%d deg F = %d deg C\n", 32, fahrenheitToCelsius(32));
}
```

Note that nameless classes (or the methods contained within) cannot be referenced from outside the definition of the class, nor can we write and execute JUnit tests. So, the utility of nameless classes is little, in our opinion.

9.2 Pattern Matching

Pattern matching is a powerful tool for working with data. It allows the programmer to create temporary bindings for identifiers that match a given pattern. This is useful for extracting data from a data structure, or for testing whether a data structure matches a given pattern. Java added support for pattern matching inside `switch` expressions in Java 21. Prior to this version, the best that could be done was to use `instanceof` to test whether an object was an instance of a given class or interface, and then cast the object to that type. Pattern matching is significantly more concise.

Example 9.1. Suppose we want to write a method that uses pattern matching to compute the perimeter of an `IShape`. We can do this by matching on the shape and then computing the perimeter for each type of shape.

Listing 9.5—Example of Pattern Matching

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

class PatternMatchingTester {

  @Test
  void patternMatchingTest() {
    IShape circle = new Circle(5);
    IShape rectangle = new Rectangle(5, 10);
    IShape triangle = new Triangle(5);

    assertAll(
      () -> assertEquals(31.41592653589793, perimeter(circle)),
      () -> assertEquals(30, perimeter(rectangle)),
      () -> assertEquals(15, perimeter(triangle)));
  }
}
```

The definitions of `Rectangle`, `Circle`, `Triangle`, and `IShape` are trivial and have been shown in previous chapters. The `perimeter` method, which is static inside `PatternMatching`, is shown below. We return the result of a `switch` expression, which matches against the possible subtypes of `IShape`. We create a temporary binding for the identifier `shape` that is bound to the `IShape` object passed into the method. This, in effect, casts the `IShape` to the subtype that is pattern matched, and we can then access the respective public methods and fields of the specific subtype rather than being restricted to only members of the `IShape` interface.

Listing 9.6—Pattern Matching

```
imoprt java.lang.IllegalArgumentException;

class PatternMatching {

  /**
   * Computes the perimeter of a given shape.
   * @param shape the IShape whose perimeter to compute.
   * @return the perimeter of the shape.
   */
  public static double perimeter(IShape s) {
    return switch (s) {
      case Rectangle r -> 2 * r.getWidth() + 2 * r.getHeight();
      case Circle c -> 2 * Math.PI * c.getRadius();
      case Triangle t -> 3 * t.getSideLength();
      default -> throw new IllegalArgumentException("perimeter: bad shape " + s);
    };
  }
}
```

We can also use "guard expressions" when constructing patterns to only match a pattern if a condition holds for that pattern.

Example 9.2. Suppose that we want to write a factorial method using pattern matching. We can do this by matching on the argument to the factorial method. If the argument is zero, we return one. Otherwise, we return the argument multiplied by the factorial of the argument minus one. We can use a guard expression to ensure that the argument is non-negative.

Listing 9.7—Example of Guard Expressions

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

class FactorialTester {

  @Test
  void factorialTest() {
    assertAll(
      () -> assertEquals(1, factorial(0)),
      () -> assertEquals(1, factorial(1)),
      () -> assertEquals(2, factorial(2)),
      () -> assertEquals(6, factorial(3)),
      () -> assertEquals(24, factorial(4)),
      () -> assertEquals(120, factorial(5)));
  }
}
```

Listing 9.8—Factorial

```
import java.lang.IllegalStateException;

class Factorial {

  public static Integer fact(Integer n) {
    return switch (n) {
      case Integer v when v == 0 -> 1;
      case Integer v when v > 0 -> v * fact(v - 1);
```

```
      default -> throw new IllegalStateException("fact: unexpected value " + n);
    };
  }
}
```

Notice that we have to use the wrapper class, since Java permits only special types of pattern matching over the primitive data types, and otherwise works with objects. As an example, we could replace the first case with a match against the literal zero, since Java autounboxes the `Integer` object to a primitive `int`, e.g., `case 0 -> 1;`. Unfortunately we cannot use guard expressions over primitives.

Pattern matching is not restricted to switch-case expressions. We can also use pattern matching in cases where we check the instance of an object, namely in the `equals` method. As an example, traditional `equals` methods look like the following:

```
@Override
public boolean equals(Object o) {
  if (o instanceof MyClass) {
    MyClass other = (MyClass) o;
    return this.field1.equals(other.field1) &&
           this.field2.equals(other.field2) &&
           ...
           this.fieldN.equals(other.fieldN);
  } else { return false; }
}
```

The need to cast the object to the type of the class on its own separate line is tedious. Pattern matching allows us to write the `equals` method in a more concise manner by providing an identifier to the pattern that is bound to the object being tested. We can wrap this in either an `if` statement or as a logical AND, because the pattern matching expression returns a boolean and fails to match if the object is not an instance of the stated class.

```
@Override
public boolean equals(Object o) {
  return (o instanceof MyClass other)     &&
         this.field1.equals(other.field1) &&
         this.field2.equals(other.field2) &&
         ...
         this.fieldN.equals(other.fieldN);
}
```

Example 9.3. Suppose that we want to override the `equals` method inside `Rectangle` using pattern matching. It's possible to use an `if` statement rather than resolving the `instanceof` check in an expression, but the latter is much more concise and solves the same problem.

Listing 9.9—Rectangle Tester

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

class RectangleTester {

  @Test
  void equalsTest() {
    Rectangle r1 = new Rectangle(5, 10);
```

```
    Rectangle r2 = new Rectangle(5, 10);
    Rectangle r3 = new Rectangle(10, 5);

    assertAll(
      () -> assertEquals(r1, r2),
      () -> assertNotEquals(r1, r3),
      () -> assertNotEquals(r1, "Hello!"));
  }
}
```

Listing 9.10—Rectangle

```
class Rectangle implements IShape {

  private final double WIDTH;
  private final double HEIGHT;

  public Rectangle(double width, double height) {
    this.WIDTH = width;
    this.HEIGHT = height;
  }

  @Override
  public boolean equals(Object o) {
    return (o instanceof Rectangle r) &&
           this.WIDTH == r.WIDTH &&
           this.HEIGHT == r.HEIGHT;
  }

  public double getWidth() { return this.WIDTH; }

  public double getHeight() { return this.HEIGHT; }
}
```

Record Types

Our perimeter pattern matcher is helpful and convenient, but what is not-so-convenient is that we have to write a lot of boilerplate code to design, for example, a class that represents a rectangle. Each subtype of `IShape` needs instance variables, a constructor, and accessors at a minimum. Moreover, we need to manually extract the fields from the object that is bound by the variable binding. In modern Java, we can utilize *record types*, which are immutable classes whose boilerplate code (as described previously) is generated by the compiler. We can also add our own methods to the record type, but we cannot add instance or static variables.

Example 9.4. Using record types, let's write a small interpreter for a simple language that supports arithmetic expressions and boolean expressions. Instead of the interface approach that we took before, we will use records and pattern matching to evaluate the expressions. With this, our interpreter will make use of `var`: a new keyword for a variable whose type is inferred from the type of the expression on the right-hand side of the assignment operator. This allows us to omit long and complex types such as `BigDecimal`.

Listing 9.11—Testing the Evaluator

```
import static Assertions.assertAll;
import static Assertions.assertEquals;
import java.lang.BigDecimal;
```

```
class EvaluatorTest {

  @Test
  void eval() {
    assertAll(
      () -> assertEquals(new BigDecimal("42"),
                         Evaluator.eval(new Number(42))),
      () -> assertEquals(new BigDecimal("42"),
                         Evaluator.eval(new Add(new Number(41), new Number(1)))));
  }
}
```

Listing 9.12—Expression Interface

```
interface IExpr {}
```

Listing 9.13—"Number" Record Type

```
import java.lang.BigDecimal;

record Number(BigDecimal value) implements IExpr {

  public Number(int value) { this(new BigDecimal(value)); }
}
```

Listing 9.14—"Add" Record Type

```
record Add(IExpr left, IExpr right) implements IExpr {}
```

Listing 9.15—Evaluator

```
import java.lang.IllegalArgumentException;
import java.lang.BigDecimal;

class Evaluator {

  /**
   * Evaluates a given expression using records and pattern matching.
   * @param exp the expression to evaluate as a subtype IExpr.
   * @return result of evaluating the expression: a BigDecimal.
   */
  public static BigDecimal eval(IExpr exp) {
    return switch (exp) {
      case Number(var n) -> n;
      case Add (var left, var right) -> eval(left).add(eval(right));
      default -> throw new IllegalArgumentException("eval: bad expr " + exp);
    }
  }
}
```

In the interpreter, we pattern match on the constructors of our record types. For example, in the `eval` method, when pattern matching on the `Add` constructor. We create temporary bindings for the identifiers `left` and `right` that are bound to the respective `Expr` objects that are passed into the constructor. We then recursively call `eval` on these objects and add the results together. We do the same for the other constructors of `Expr`. To make testing the program easier, we designed a secondary constructor for `Number` that receives an integer and converts it to a `BigDecimal`. Note that self-defined constructors of a record can only reference other constructors via `this(...)`.

Example 9.5. Let's use some of Java's modern features to write a lexer and parser for the interpreter that we designed in Chapter 5. We presented this as an exercise to the readers at the end, but we imagine it was quite a hassle, if the readers restrict themselves to older versions of Java. In particular, we can use of records and pattern matching to greatly reduce the amount of necessary boilerplate code. Our language capabilities will be greatly reduced for the time being, however, since our focus is the lexing and parsing of the raw language rather than its evaluation. Therefore, let's restrict the language to supporting numbers, symbols, and s-expressions, i.e., expressions enclosed by parentheses. Below we present some examples of strings within this language.

```
7
(+ 2 40)
(+ 10 (* 4 5) 30)
```

For the uninformed, a *lexer* is a program that converts input into *tokens*, also called a tokenizer. Tokens are components of an input string. For example, we might convert the string `"(+ 2 3)"` into five tokens:

```
L_PAREN "("
SYMBOL "+"
NUMBER "2"
NUMBER "40"
R_PAREN ")"
```

It is the job of the lexer to categorize the input into patterns that the programmer ascribes, e.g., `NUMBER` or `SYMBOL`. Our lexer will assume tokens are separated by whitespace, which for the most part, this holds true. Consider the simplest case of applying an operator to a list of operands: `"(+ 2 40)"`. Should we split this example string on the space character, we accidentally group `"(+"`, an undesired token. The solution is to add a whitespace after each opening parenthesis and a whitespace before each closing parenthesis.

Before we write the lexer and test cases, we need to determine what comprises a token. Tokens have token types, e.g., `L_PAREN` and so forth, as well as an associated string. In the case of, say, a `NUMBER`, the associated string *is* that number represented by the token, e.g,. `40`. Let's use a Java record to represent a `Token`, and an enumeration for the types of tokens. We will integrate tokens into the lexer tests, so it makes little sense to write separate tests for tokens and token types.

Listing 9.16—Token Type Enumeration

```java
enum TokenType { L_PAREN, R_PAREN, NUMBER, SYMBOL }
```

Listing 9.17—Token Record

```java
record Token(TokenType type, String data) { }
```

The implementation of the lexer is not complicated—the static `lex` method receive a `String` and returns a `Queue<Token>` containing all identified tokens. As we stated earlier, we first add the required spacing, then split on the whitespace character, producing a `String[]` of raw tokens. Then, we iterate over these strings and enqueue an instance of `Token` with the respective token type and data. For identifying numbers, we simply determine if an attempt to parse the string as a number results in an exception and, if not, it becomes a NUMBER token. Otherwise, the only option (assuming we identify parentheses before this case) is some kind of symbol token.

Listing 9.18—Testing the Lexer

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

import java.util.LinkedList;
import java.util.List;

class LexerTester {

  @Test
  void testLex() {
    String in1 = "42";
    String in2 = "(+ 2 40)";
    String in3 = "(+ 27 (* 5 3) 10)";
    assertAll(
      () -> assertEquals(new LinkedList<>(
                          List.of(new Token(TokenType.NUMBER, "42"))),
                         Lexer.lex(in1)),
      () -> assertEquals(new LinkedList<>(
                          List.of(new Token(TokenType.L_PAREN, "("),
                                  new Token(TokenType.SYMBOL, "+"),
                                  new Token(TokenType.NUMBER, "2"),
                                  new Token(TokenType.NUMBER, "40"),
                                  new Token(TokenType.R_PAREN, ")"))),
                         Lexer.lex(in2)),
      () -> assertEquals(new LinkedList<>(
                          List.of(new Token(TokenType.L_PAREN, "("),
                                  new Token(TokenType.SYMBOL, "+"),
                                  new Token(TokenType.NUMBER, "27"),
                                  new Token(TokenType.L_PAREN, "("),
                                  new Token(TokenType.SYMBOL, "*"),
                                  new Token(TokenType.NUMBER, "5"),
                                  new Token(TokenType.NUMBER, "3"),
                                  new Token(TokenType.R_PAREN, ")"),
                                  new Token(TokenType.NUMBER, "10"),
                                  new Token(TokenType.R_PAREN, ")"))),
                         Lexer.lex(in3)));
  }
}
```

Listing 9.19—Implementation of Lexer

```
import java.util.LinkedList;
import java.util.Queue;

class Lexer {

  /**
   * Splits the input of our language into tokens.
   * @param raw string input.
   * @return queue of tokens.
   */
  public static Queue<Token> lex(String s) {
    // First, we need to add a space after each left parenthesis and before each
        right parenthesis.
```

```java
    String sWithSpaces = s.replaceAll("\\(", "( ").replaceAll("\\)", " )");
    // First, we need to split the string into tokens.
    String[] rawTokens = sWithSpaces.split(" ");
    return Arrays.stream(rawTokens)
                 .map(Lexer::createToken)
                 .collect(Collectors.toCollection(LinkedList::new));
  }

  /**
   * Creates a token from a string.
   * @param t the string to create a token from.
   * @return the token.
   */
  private static Token createToken(String t) {
    return switch (t) {
      case "(" -> new Token(TokenType.L_PAREN, t);
      case ")" -> new Token(TokenType.R_PAREN, t);
      default -> {
        try {
          Double.parseDouble(t);
          yield new Token(TokenType.NUMBER, t);
        } catch (NumberFormatException ex) {
          yield new Token(TokenType.SYMBOL, t);
        }
      }
    };
  }
}
```

As we see, we can make use of streams and collectors to populate the queue of tokens. The `createToken` method is responsible for creating the respective token type from the input. Subsequent updates to the grammar of the language would result in only needing to update `createToken` rather than requiring a change to the internals of the lexer algorithm itself. Suppose that we want to add boolean literals into the language. Thus, we need two modifications: 1) add a boolean `TokenType` and 2) add a clause in the switch expression to return a `Token` that represents a `TokenType.BOOLEAN` or something similar.

Example 9.6. We will break the mini-project into two separate examples since both the lexer and parser are complicated pieces of the puzzle. The parser receives tokens and, in general, creates an abstract syntax tree to represent the input data.[1]

Our parser implementation contains a single static method: `parse`, which receives a queue of tokens and returns the root of an abstract syntax tree. We need to create three kinds of abstract syntax trees: `NumNode`, `SymbolNode`, and `SExprNode`, all of which extend the abstract `AstNode` class. An `AstNode`, like the one presented from Chapter 5, contains a list of children, as well as a way to add children to that list. The former two subclasses, namely `NumNode` and `SymbolNode`, store their data, i.e., the number and symbol respectively, as instance variables inside their class definitions. Because we have seen these two class definitions previously, we will omit their implementation. What we have not seen before, however, is `SExprNode`. Because our language is so small, an `SExprNode` is identical to an `AstNode` aside from the fact that it is not abstract.

To parse the list of tokens, we peek at the head of the queue and match on its type. If it is a number or a symbol, we instantiate an instance of the appropriate subclass type. If we

[1] We say "in general" because a parser might also forego an AST and go straight to an interpreter or program instructions.

encounter a left parenthesis, then we must do more work, that work being to parse the contents inside the parentheses. We traverse over the tokens until we encounter a right parenthesis, recursively parsing and adding each abstract syntax tree node to a running `SExprNode` instance. Because the process is recursive, we handle the cases in which an s-expression is nested inside another, which was the case in the third test case from the lexer tester.

Listing 9.20—Testing the Parser

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

class ParserTester {

  @Test
  void testParse() {
    String in1 = "42";
    String in2 = "(+ 2 40)";
    String in3 = "(+ 27 (* 5 3) 10)";

    assertAll(
      () -> assertEquals(new NumNode(42),
                         Parser.parse(Lexer.lex(in1))),
      () -> assertEquals(new SExprNode(
                          new SymbolNode("+"),
                          new NumNode(2),
                          new NumNode),
                         Parser.parse(Lexer.lex(in2))),
      () -> assertEquals(new SExprNode(
                          new SymbolNode("+"),
                          new NumNode(27),
                          new SExprNode(
                           new SymbolNode("*"),
                           new NumNode(5),
                           new NumNode(3)),
                          new NumNode(10)),
                         Parser.parse(Lexer.lex(in3))));
  }
}
```

Listing 9.21—Parser Implementation

```java
import java.util.Objects;
import java.util.Queue;

class Parser {

  /**
   * Parses a list of tokens into an AST.
   * @param tokenList queue of tokens to parse.
   * @return constructed AST from the tokens.
   */
  public static AstNode parse(Queue<Token> tokenList) {
    if (!tokenList.isEmpty()) {
      return switch (tokenList.peek().type()) {
        case NUMBER -> new NumberNode(tokenList.remove().data());
        case SYMBOL -> new SymbolNode(tokenList.remove().data());
        case BOOLEAN -> new BooleanNode(tokenList.remove().data());
        case L_PAREN -> parseSExpression(tokenList);
```

```java
          default -> throw new IllegalArgumentException("parse: unexpected token "
                                                   + tokenList.peek().type());
        };
      } else { return null; }
    }

    /**
     * Constructs an s-expression from a list of tokens. The precondition
     * for entering this method is that the first token in the queue is
     * the opening parenthesis of the s-expression, i.e., an L_PAREN.
     * @param tokenList queue of tokens to parse as the sexpr.
     * @return constructed SExpressionNode from the tokens.
     */
    private static AstNode parseSExpression(Queue<Token> tokenList) {
      if (tokenList.isEmpty()) { return null; }
      else {
        // Remove the left parenthesis.
        tokenList.remove();
        SExpressionNode sexpr = null;

        // Parse the tokens until we reach the right parenthesis.
        while (!tokenList.isEmpty()
            && tokenList.peek().type() != TokenType.R_PAREN) {
          // Get the first token.
          sexpr = sexpr == null ? parseFirstToken(tokenList) : sexpr;
          sexpr.addChild(parse(tokenList));
        }

        // Remove the right parenthesis.
        tokenList.remove();
        return sexpr;
      }
    }

    /**
     * Instantiates a subclass of SExpressionNode to the type defined by the first
     * token in the queue. This can be anything represented as an s-expression.
     */
    private static SExpressionNode parseFirstToken(Queue<Token> tokenList) {
      switch (Objects.requireNonNull(tokenList.peek()).type()) {
        default: { return new SExpressionNode(); }
      }
    }
  }
```

Fortunately the parser is also not to difficult to understand. We perform a case analysis on the token and either return an abstract syntax tree node or create one from an s-expression.

We decided to write a helper method, `parseFirstToken`, as a precursor to more advanced features in the language. Many special forms are represented as s-expressions and should be converted into appropriate abstract syntax tree nodes. As an example, we can write a conditional expression, i.e., `(cond pred cons alt)` via an s-expression. Therefore it makes logical sense to have a separate method that reads the first token and instantiates the correct node for that token, whether it be a `CondNode` or something else. Our implementation simply has one case, `default`, since we do not have conditional expressions, but we encourage the readers to add them as language features!

9.3 Reflection

Reflection, while not a necessarily new computer science concept, is a powerful way for programming languages to interpret and potentially modify its own structure. We made an example of reflection in our chapter on classes and objects to pass around a class type as a parameter.

Example 9.7. Suppose that we want to be able to search, then invoke, a method based on its name. Reconsider the primitive calculator example from many chapters ago, where we utilize a case dispatch on the operator received as a terminal argument. As we add functions to the system, we must proportionally add code in the `main` method to account for the new case, which is tiresome at best. Java allows us to lookup a method, at runtime, using its reflection API, and pass parameters accordingly. As a substitute for terminal arguments (and the `main` method in general), we will pass an "argument array" to a static `calculate` method, so we can easily run unit tests.

Listing 9.22—Reflective Calculator Tester

```java
import static Assertions.assertAll;
import static Assertions.assertEquals;
import static ReflectiveCalculator.calculate;

class ReflectiveCalculatorTester {

  private static final double DELTA = 0.0001;

  @Test
  void testCalculator() {
    assertAll(
      () -> assertEquals(5,
                         calculate(new String[] {"add", "2", "3"}),
                         DELTA),
      () -> assertEquals(5,
                         calculate(new String[] {"subtract", "10", "5"}),
                         DELTA),
      () -> assertEquals(10,
                         calculate(new String[] {"multiply", "2", "5"}),
                         DELTA),
      () -> assertEquals(2,
                         calculate(new String[] {"divide", "10", "5"}),
                         DELTA));
  }
}
```

To retrieve a method at runtime, we first need to retrieve the class in which it lives through the `Class.forName` method. In this case we pass `ReflectiveCalculator`, as that is the name of our class. Note that the return value is of type `Class<?>`, denoting that it is a reflective class type.

With the class type object in hand, we must now get the method object to call. To do so we need two values: its identifier, and its parameter types. We will assume that the arguments to the calculator functions are strings, which can be converted inside the respective methods. This approach makes it significantly easier to inform the reflection API of our parameter type(s). Therefore we use the `getDeclaredMethod` method, which takes the name of the method as a string, and a variadic number of `Class<?>` objects that represent the parameter types. In our case, the parameter type is a `String[]`, so we pass `String[].class`. Because we might attempt to reference a non-existent method, we must wrap this in a `try-catch` block.

Finally, we invoke the method encapsulated by the `Method` using the conveniently named `invoke` method, which receives the object on which to invoke the method (in this case, `null` because the method is static), and the necessary arguments. The return value is an `Object`, so we must cast it to the appropriate type. There are several issues that may arise when calling a method, such as the method being inaccessible due to its access modifier, the method throwing a checked exception, or passing the wrong number of arguments. We must handle these issues accordingly via `try-catch` blocks.[1] The four calculator methods are trivial and have been omitted; all they do is convert the string arguments to numbers and perform, then return, the corresponding operation.

There is one small intricacy with how Java handles variadic arguments and passing arrays to reflective methods. Passing an array to a variadic argument method unwraps the arguments and passes them individually. For example, if we have a method `foo(String... args)`, and we pass `foo(new String[] {"Hello", "World"})`, the method receives two arguments: `"Hello"` and `"World"`. In our case this is problematic, since we want our computation methods to receive the entire array of arguments. To accomplish this, we can cast the array to an `Object` and pass it to `invoke` as a single argument.

Listing 9.23—Reflective Calculator

```java
import java.lang.reflect.Method;
class ReflectiveCalculator {

  /**
   * Calculates the result of a given operation on two numbers.
   * @param args the operation to perform, and the two numbers.
   * @return the result of the operation.
   */
  static double calculate(String[] args) {
    Class<?> cls = ReflectiveCalculator.class;
    try {
      Method mtd = cls.getMethod(args[0], String[].class);
      return (double) mtd.invoke(null, args);
    } catch (NoSuchMethodException
           | InvocationTargetException
           | IllegalAccessException ex) {
      throw new RuntimeException(ex);
    }
  }
}
```

Example 9.8. The Python programming language, including many others, offer a REPL, or a read-evaluate-print-loop, that users can run at the terminal to evaluate expressions and programs. This is done to alleviate the need to open a source file in a text editor, type the code, then run the file with the respective command. Java is not one of these languages out-of-the-box, unfortunately. There is an application, namely JShell, which introduces

[1] Our code uses the Java 7 feature of the vertical pipe '|' to catch/handle multiple exceptions at once, removing duplicate code.

this functionality. In this example, we will write our own version of JShell, where the programmer can type and evaluate expressions and statements at the terminal.[1]

First, we need to establish a few details. Our program will read, from standard input, a subset of Java code. After reading, we spin up an instance of the Java compiler and compile the source code into a `.class` file. Finally, using reflection, we search for and execute that code. This is the goal at a high level, but we certainly need to break this down into sub-components. Let's see some examples of what we will be able to do in due time:

```
> int i = 5;
> System.out.println(i)
5
> class Animal { String speak() { return ""; }}
> class Dog extends Animal { String speak() { return "Wolf!"; }}
> Animal a = new Dog();
> System.out.println(a.speak());
"Wolf!"
> System.out.println(List.of(i, i + 2));
[5, 7]
> System.out.println(IntStream.iterate(0, i -> i + 5).limit(10));
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

As we can see, powerful programs are a possibility only by using the REPL. The question now is, how do we get to that point?

The idea with this project is to output our lines of source code into a temporary Java class. In essence, we create a temporary directory on the system that that stores a separate class (file), enumerated from zero, containing whatever we enter at the REPL. Each time we enter code, we wrap it inside a class called `"InterpX"` where `X` is the current class identifier, so to speak. These classes are extended with each new segment of code received to allow access to prior declarations/definitions. As an example, consider the following class definitions generated after inputting the first two lines that we showed earlier:

```
class Interp0 {
  public static int i = 5;
  public static void execute() {}
}
class Interp1 extends Interp0 {
  public static void execute() {
    System.out.println(i);
  }
}
```

The `execute` method is generated all the time, but we only care about its contents (i.e., we only populate its body) when we write something that is not a declaration. For example, if we write a class definition or a variable, it is nonsensical to place it inside the `execute` method body, since in the former case it would not compile, and in the latter case it would be local to only that version of `execute`. So, we need a way of determining whether or not a piece of code *is* a declaration/definition. All declarations in our subset of Java will be restricted to the following grammar:

[1] Credit goes to Terence Parr for this example and "assignment" from his graduate course on programming languages at the University of San Francisco. That's right–this is a graduate-level programming example!

⟨*expr*⟩	::=	⟨*varDecl*⟩
	|	⟨*methodDecl*⟩
	|	⟨*classDecl*⟩
⟨*varDecl*⟩	::=	⟨*type*⟩ ‘ ’ ⟨*id*⟩ ‘;’
	|	⟨*type*⟩ ‘[]’ ‘ ’ ⟨*id*⟩ ‘;’
⟨*methodDecl*⟩	::=	⟨*type*⟩ ‘ ’ ⟨*id*⟩ ‘(’ ⟨*params*⟩ ‘)’ ‘{’ ⟨*body*⟩ ‘}’
⟨*classDecl*⟩	::=	‘`class` ’ ⟨*id*⟩ ‘{’ ⟨*body*⟩ ‘}’
	|	‘`class` ’ ⟨*id*⟩ ‘ `extends` ’ ⟨*id*⟩ ‘{’ ⟨*body*⟩ ‘}’

Figure 9.1: Extended BNF Grammar for Declarations

Let's assume that the `type` rule is any arbitrary sequence of characters representing a type in Java (e.g., `void`, `int`, `Dog`), as is an identifier. Under these assumptions, it is simple to parse a string to determine if it's a declaration. We will further assume that the user is acting in good faith and typing only valid Java code that compiles under said assumptions; responses to inputs such as `inttttt x = 5;` will not be considered in this example.[1]

Listing 9.24—Declaration Determination Methods

```java
class JavaRepl {

  private boolean isDeclaration(String s) {
    return this.isVariableDeclaration(s)
        || this.isMethodDeclaration(s)
        || this.isClassDeclaration(s);
  }

  /**
   * Determines if a given string is a type of Java variable declaration.
   * Types of var declarations:
   * - int x;
   * - int x = 5;
   * - int[] a = new int[5];
   * - String x = "Hi!";
   * - ArrayList<Integer> foo;
   */
  private boolean isVariableDeclaration(String s) {
    return s.matches("[a-zA-Z_\\[\\]<>]+\\s*[a-zA-Z0-9_]+(\\s*=\\s*.+?)?;");
  }

  /**
   * Determines if a given string is a type of Java method declaration.
   * Types of method declarations:
   * - int foo() {}
   * - int bar(int x, int y) {}
   * - String baz(ArrayList<Integer> foo) {}
```

[1] We make prolific use of regular expressions in this next section. You are free to gloss over these details or use raw string parsing with substring and equals, but regex makes the parsing process significantly faster.

```
   */
  private boolean isMethodDeclaration(String s) {
    return s.matches("[a-zA-Z_\\[\\]<>]+\\s+[a-zA-Z0-9_]+\\(.*?\\)\\s*\\{.*?\\}");
  }

  /**
   * Determines if a given string is a type of Java class declaration.
   * Types of class declarations:
   * - class Animal {}
   * - class Dog extends Animal {}
   */
  private boolean isClassDeclaration(String s) {
    return s.matches("class\\s+[a-zA-Z_<>]+\\s+\\{.*?\\}")
        || s.matches("class\\s+[a-zA-Z_<>]+\\s+extends\\s+[a-zA-Z_<>]+\\s
                     *\\{.*?\\}");
  }
}
```

Anything that is not a declaration is something else that belongs inside the body of `execute`, and is executed accordingly. For instance, if we declare i to be zero, then use a post-increment operator on i, the `i++` statement is stored inside `Interp1`'s `execute` body, and is invoked prior to the next standard input reading.

Now, onto the main event. We know how to read lines/data from standard input, so that's nothing more than a rehashing of what we've seen before. We create an infinite loop, read in a single string (line), and create a `File` object out of that string by storing it in a temporary class file. Where do we create that temporary class file? In a temporary directory, which is created by a static method in the`Files` class. Let's set this up in the constructor of `JavaRepl`, since the temporary directory (path) won't change for the lifetime of the program. Let's also instantiate the standard input reading mechanism, i.e., a `BufferedReader` that operates over an `InputStreamReader`.

Listing 9.25

```
import java.io.*;
import java.nio.file.Files;
import java.nio.file.Path;

class JavaRepl {

  private final Path TMP_DIR;

  private final BufferedReader IN;

  private int classNo;

  public JavaRepl() {
    try {
      this.TMP_DIR = Files.createTempDirectory();
    } catch (IOException ex) {
      throw new RuntimeException(ex);
    }

    this.IN = new BufferedReader(new InputStreamReader(System.in));
    this.classNo = 0;
  }
}
```

With this, we know that we want to store the class and what its name should be, thanks to a counter that starts at zero. In the `createJavaFile` method, we create a `File` at the correct location, with its contents populated by a writer of some kind. Here is where we make use of the `isDeclaration` method—if the source code that we pass is, in fact, a declaration, we precede it with the `public static` keywords. Otherwise, it resides inside the `public static void execute()` method. Creating and returning this file is straightforward. We need to account for the fact that if the class number is zero, we do not extend any class. As a corollary point, to be able to use certain classes, e.g., `List`, at the REPL, we include two wildcard imports from the I/O and utilities Java packages.

Listing 9.26

```java
class JavaRepl {
  // ...  other methods not shown.

  private File createJavaFile(String src) {
    File f = new File(String.format("%s/Interp%d.java",
                                    this.TMP_DIR,
                                    this.classNo));
    try (PrintWriter pw = new PrintWriter(new FileWriter(f))) {
      pw.println("import java.io.*;");
      pw.println("import java.util.*;");
      // If it is the starting class we do not have it extend anything.
      pw.println(String.format("class Interp%d %s", this.classNo,
                               this.classNo == 0
                               ? "{"
                               : "extends Interp" + (this.classNo - 1) + " {"));

      // If it's a declaration, it cannot be inside a method.
      if (this.isDeclaration(src)) {
        pw.printf("public static %s\n", src);
        pw.println("public static void execute() {}");
      } else {
        pw.printf("public static void execute() { %s }\n", src);
      }
      pw.println("}");
    } catch (IOException ex) {
      ex.printStackTrace();
    }
    return f;
  }
}
```

Here we run into our first of two roadblocks: how to compile the file at runtime. Fortunately, there exists the Java compiler API, which contains methods and classes to compile a file definition. Most of this code is boilerplate "setup," so understanding it completely is unnecessary. At the core, we instantiate a compiler object, as well as a list of diagnostics that may result from compiling the program (e.g., error messages). These get passed into a compilation unit that executes the compiler. Upon success, it produces a `.class` file in the temporary directory, as we would if we were compiling our Java code by hand. We will instantiate the `JavaCompiler`, `StandardJavaFileManager`, and `DiagnosticCollector` objects in the constructor. More importantly we have the `executeRepl` method, which starts the REPL and listens for data on standard input. This is where we use the API to compile and produce the `.class` file.

Listing 9.27—Using the Java Compiler API

```java
import javax.tools.*;
import java.lang.StringBuilder;

class JavaRepl {

  // ...  other instance variables not shown.

  private final JavaCompiler JAVAC;

  private final StandardJavaFileManager FILE_MANAGER;

  private final DiagnosticCollector<? super JavaFileObject> DS;

  public JavaRepl() {
    // ...  other instantiations not shown.
    this.JAVAC = ToolProvider.getSystemJavaCompiler();
    this.DS = new DiagnosticCollector<>();
    this.FILE_MANAGER = this.JAVAC.getStandardFileManager(this.DS, null, null);
  }

  /**
   * Continuously loops, reading in lines of input representing valid Java
   * programs. These are converted into statements/expressions that are fed into
   * a skeleton Java class file. We use Java's runtime compiler to execute
   * these, and the reflection API to dynamically load the class at runtime.
   */
  private void executeRepl() throws Exception {
    List<File> programs = new ArrayList<>();

    while (true) {
      // For now assume that one line is the entire program.
      StringBuilder sb = new StringBuilder();
      System.out.print("> ");
      sb.append(this.IN.readLine());
      programs.add(this.createJavaFile(sb.toString()));

      // Create the compiler from these files.
      var units = this.FILE_MANAGER.getJavaFileObjectsFromFiles(programs);
      this.task = (JavacTask) this.JAVAC.getTask(null, this.FILE_MANAGER, this.DS,
                                                  null, null, units);

      // Compile the list of programs.
      boolean ok = this.task.call();
      if (!ok) {
        for (var diag : this.DS.getDiagnostics()) {
          System.err.println(diag);
        }
      } else {
        // TODO.
      }
    }
  }
}
```

If the compilation was unsuccessful, thereby meaning `ok` is false, we iterate over the diagnostics received from the compile and print them to the standard error stream. Otherwise, we know it successfully compiled and we now have a class file.

Now comes the second roadblock: we want to load this file into memory via the reflection API and preserve changes to static fields. For example, if we increment/reassign a variable, its state should be updated across the REPL. To do this, we need to use a common class loader, which persists changes to static fields and means we can load a new class at runtime. Indeed, we want to load the new class that we just compiled, namely `Interp`X, and invoke its `execute` method. This is nothing new, but what we have not seen is the common class loader approach; we store an instance of `URLClassLoader` that gets instantiated to refer to the temporary directory path, in our constructor. Therefore, when loading a class via `loadClass`, any changes we made to previous variables remain loaded into memory.

Listing 9.28—Using the Reflection API

```java
class JavaRepl {

  // ... other instance variables not shown.

  private final URLClassLoader CLASS_LOADER;

  public JavaRepl() {
    // ... other instantiations not shown.
    try {
      this.CLASS_LOADER = URLClassLoader.newInstance(
                              new URL[]{this.TMP_DIR.toUri().toURL()});
    } catch (MalformedURLException e) {
      throw new RuntimeException(e);
    }
  }

  /**
   * Continuously loops, reading in lines of input representing valid Java
   * programs. These are converted into statements/expressions that are
   * fed into a skeleton Java class file. We use Java's runtime compiler to
   * execute these, and the reflection API to dynamically load the class at
   *      runtime.
   */
  private void executeRepl() throws Exception {
    List<File> programs = new ArrayList<>();

    while (true) {
      // ... compilation omitted.
      if (!ok) {
        // ... omitted.
      } else {
        Class<?> cls = CLASS_LOADER.loadClass("Interpret" + this.classNo);
        Method method = cls.getMethod("execute");
        method.invoke(null);
        this.classNo++;
      }
    }
  }
}
```

In summary, we load the current class, reflectively grab its `execute` method, then execute it with `null` as an argument, designating that it is a static method in the class. Finally we increment the class number for the next line.

9.4 Concurrent Programming

There is sometimes a conflation between concurrency and parallelism, two related but different methodologies. *Concurrency* describes actions/computations that occur at the same time, whereas *parallelism* refers to simultaneous actions/computations. Whereas a non-concurrent program will complete tasks $t_1, t_2, \ldots, t_n$ one after the next, concurrent programs will start t_1, then start t_2, and so forth, without necessarily finishing t_1 before starting subsequent tasks. The order of execution and completion depends on the operating system's scheduler, but what is important is that no tasks t_i and t_j operate at the exact same moment. Contrast this idea with parallelism, which does allow tasks t_i and t_j to be worked on at the exact same moment.

We can simulate the concurrency/parallelism distinction with two queues of people A and B in line for a ride at an amusement park. A concurrent system might poll a person from A, then from B, then back to A, but never from A and B at the same time. On the other hand, consider a kiosk that has multiple cashiers serving several people simultaneously, polling from both A and B. The kiosk system is, therefore, operating in parallel.

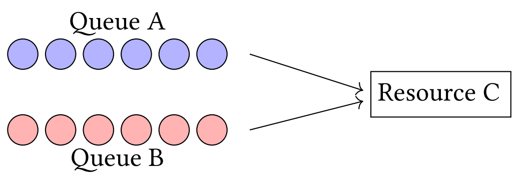


Figure 9.2: Diagram of Concurrency

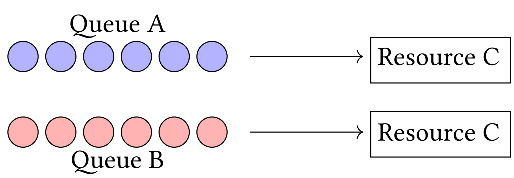


Figure 9.3: Diagram of Parallelism

Threads

Java supports concurrent programming through its Thread API, but what is a thread? In essence, a *thread* is a single lightweight process that executes a task, or a sequential set of tasks.

Example 9.9. Multiple threads may access shared data at the same time, which is convenient. The problem is the fact that threads can *context switch* at arbitrary points, which may include during an *atomic operation*. Atomic operations are operations that must be executed in their entirety, without interruption. For example, suppose that we have a variable `counter` that we want to increment. We might write the following code:

Listing 9.29—Example of a Race Condition

```java
import java.lang.Thread;
import java.lang.Runnable;

class RaceConditionExample {

  private static int counter = 0;

  public static void main(String[] args) {
    Thread t1 = new Thread(new Incrementer());
    Thread t2 = new Thread(new Incrementer());

    t1.start();
    t2.start();

    try {
      t1.join();
      t2.join();
    } catch (InterruptedException ex) { ex.printStackTrace(); }

    System.out.println(counter);
  }

  private static class Incrementer implements Runnable {

    @Override
    public void run() {
      for (int i = 0; i < 1_000_000; i++) { counter += 1; }
    }
  }
}
```

Upon running the program, we might expect the output to be `2_000_000`, but this is not the case. The output is nondeterministic, meaning that it is not guaranteed to be the same every time we run the program. This is because the `counter += 1` operation is not atomic. That is, the following sequence of events can occur:

1. Thread t_1 reads the value of `counter` as `0`.
2. Thread t_2 reads the value of `counter` as `0`.
3. Thread t_1 increments the value of `counter` to `1`.
4. Thread t_2 increments the value of `counter` to `1`.
5. Thread t_1 writes the value of `counter` as `1`.
6. Thread t_2 writes the value of `counter` as `1`.

Thus, we need a way of synchronizing access to the `counter` variable. We will discuss this, alongside a more-detailed example, in the next section.

Synchronization

Example 9.10. Let's suppose that we're writing a Java class to represent a bank account, which allows users to withdraw and deposit money. Further suppose that we have multiple threads that access the same bank account at once, eah depositing then immediately withdrawing five hundred dollars.

Listing 9.30—Bank Account Class Definition

```
class BankAccount {

  private int amt;

  public BankAccount(int amt) { this.amt = amt; }

  public void deposit(int more) { this.amt += more; }

  public void withdraw(int less) {
    if (amt >= less) { this.amt -= less; }
    else { System.err.printf("withdraw: insufficient funds %d\n", less); }
  }

  public int getAmount() { return this.amt; }
}
```

Let's design a `TransactionThread` class, that implements `Runnable`, which executes one thousand "transactions," i.e., a deposit of $500, then a withdrawal of $500.[1] At the end, assuming everything works as expected, we should net zero dollars in the account.

Listing 9.31—Creating Multiple Transactions

```
import java.lang.Runnable;

class TransactionThread implements Runnable {

  private BankAccount acc;

  public TransactionThread(BankAccount acc) { this.acc = acc; }

  @Override
  public void run() {
    for (int i = 0; i < 1000; i++) {
      this.acc.deposit(500);
      this.acc.withdraw(500);
    }
  }
}
```

Listing 9.32—Running "Transaction" Threads

```
import java.lang.Thread;

class BankAccountRunner {

  public static void runTransactionThreads() {
    BankAccount acc = new BankAccount(0);
    Thread t1 = new Thread(new TransactionThread(acc));
    Thread t2 = new Thread(new TransactionThread(acc));
    Thread t3 = new Thread(new TransactionThread(acc));

    t1.start();
    t2.start();
```

[1] It is an implementer's choice to either extend the `Thread` class or implement the `Runnable` interface. We choose the latter because Java allows us to implement multiple interfaces, but we can only extend one parent class. This does, however, mean that any time we want to instantiate a thread, we must also instantiate a separate instance of the `Runnable` subtype.

```
    t3.start();

    // Wait on the threads to finish, then print the result.
    try {
      t1.join();
      t2.join();
      t3.join();
    } catch (InterruptedException ex) { ex.printStackTrace(); }

    System.out.printf("%d\n", acc.getAmount());
  }

  public static void main(String[] args) {
    runTransactionThreads();
  }
}
```

Running the program may produce the following output:

```
withdraw: insufficient funds
withdraw: insufficient funds
0
```

It seems that we have an unpredictable problem: rerunning the program produces different results each time, but the error stems from the fact that our accesses to the bank account are not *synchronized*. To understand why, let's expand out the code for performing a withdraw.

```
public void withdraw(int less) {
  int currAmt = this.amt;
  if (currAmt >= less) {
    int newAmt = currAmt - less;
    this.amt = newAmt;
  } else {
    // ... print not shown.
  }
}
```

Recall how threads work—a thread t_1 might reach the third line of this method, and store some value in `newAmt`. At this point, suppose another thread t_2 makes a deposit of $500. Finally, t_1 updates `newAmt` to zero, because `currAmt - less` must be zero, so we effectively nullify the deposit action by our second thread! As we showed with the previous example, this is a data race for the `amt` instance variable. To fix the problem, we want to ensure that our `withdraw` and `deposit` methods are synchronized, which we can do via marking the methods as `synchronized`. Synchronization of a method solves the problem of atomicity, i.e., a synchronized method can only be accessed by a single thread at any time. Let's mark our methods then rerun the code to check the result, which should never display an error. Even though the calls to `deposit` and `withdraw` might not be inside a synchronized environment, we end up netting zero in the account, because three successive calls to `deposit` must be followed, at some point, by three calls to `withdraw`, which our synchronization action guarantees.

Example 9.11. Let's design a `transfer` method as part of the `BankAccount` class, which receives another `BankAccount` b_2 and an amount n as an instance variable. We will transfer n dollars from `this` bank into b_2, which resolves to a call to `withdraw` on `this` and a call to `deposit` on b_2.

Listing 9.33—Bank Account Transfer Method Tester

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

class BankAccountTester {

  @Test
  public void testTransfer() {
    BankAccount b1 = new BankAccount(0);
    BankAccount b2 = new BankAccount(100);
    assertAll(
      () -> b2.transfer(b1, 30),
      () -> assertEquals(30, b1.getAmount()),
      () -> assertEquals(70, b2.getAmount()));
  }
}
```

Listing 9.34—Bank Account Transfer Method

```java
class BankAccount {

  private int amt;

  public BankAccount(int amt) { this.amt = amt; }

  /**
   * Transfers money from one account to another.
   * @param b2 other bank account to deposit money into.
   * @param amt the amount to withdraw from this account.
   */
  public void transfer(BankAccount b2, int amt) {
    this.withdraw(amt);
    b2.deposit(amt);
  }

  // deposit and withdraw not shown.
}
```

This code has the problem of not being synchronized; if multiple threads call `transfer` on the same bank account, we run into the same problem as before. To fix this, we may think that we need to synchronize `transfer`, but this is not the case, because a method that is synchronized means that only one thread can enter the method, but we need to synchronize the bank objects themselves from access and mutation. We can do so by synchronizing on the bank objects themselves, which we can do by using the `synchronized` keyword on a block of code. First, we synchronize on `this` (the bank account that we are withdrawing from), followed by a synchronization on `b2` (the bank account that we are depositing into). This ensures that only one thread can access both bank accounts concurrently.

Let's now create another class that implements `Runnable` to manage multiple transfers between bank accounts. Each thread will deposit \$500 into its first bank b_1, then transfer \$500 from b_1 to b_2, followed by a withdrawal of \$500 from b_2 to ensure that both accounts net zero dollars. Our tester will create two threads: one that transfers from b_1 to b_2, and another that transfers from b_2 to b_1.

Listing 9.35—Transfer Thread Implementation

```
import java.lang.Runnable;

class TransferThread implements Runnable {

  private final BankAccount B1;
  private final BankAccount B2;

  public TransferThread(BankAccount b1, BankAccount b2) {
    this.B1 = b1;
    this.B2 = b2;
  }

  @Override
  public void run() {
    for (int i = 0; i < 1000; i++) { this.B1.transfer(this.B2, 500); }
  }
}
```

Listing 9.36—Running "Transfer" Threads

```
import java.lang.Thread;

class BankAccountRunner {

  public static void runTransferThreads() {
    BankAccount b1 = new BankAccount(0);
    BankAccount b2 = new BankAccount(0);
    Thread t1 = new Thread(new TransferThread(b1, b2));
    Thread t2 = new Thread(new TransferThread(b2, b1));

    t1.start();
    t2.start();

    try {
      t1.join();
      t2.join();
    } catch (InterruptedException ex) { ex.printStackTrace(); }
  }

  public static void main(String[] args) {
    runTransferThreads();
  }
}
```

Listing 9.37—Synchronized Transfer Method

```
class BankAccount {

  private int amt;

  public BankAccount(int amt) { this.amt = amt; }

  /**
   * Transfers money from one account to another.
   * @param b2 other bank account to deposit money into.
   * @param amt the amount to withdraw from this account.
   */
  public void transfer(BankAccount b2, int amt) {
```

```
    synchronized (this) {
      synchronized (b2) {
        this.withdraw(amt);
        b2.deposit(amt);
      }
    }
  }

  // deposit and withdraw not shown.
}
```

Upon running this program it is very likely to encounter an unexpected problem: an infinite "loop," so to speak. This occurs because we have a *deadlock*, which happens when two more more threads are waiting on each other to release a lock. In our case, thread t_1 synchronizes on itself and acquires the lock. Then, suppose we context switch into t_2 and it synchronizes on itself. At this point t_1 is waiting for the lock by t_2, and t_2 is waiting on t_1's lock. There are several possible solutions to this problem, where one is to use a Java-defined `Lock` object. As we will discuss in the subsequent example, a reentrant lock allows a thread to acquire the lock multiple times, which is useful in the event that a thread needs to call a synchronized method from within a synchronized block. In particular, we will lock `transfer`, `deposit`, and `withdraw` using the same static lock. We will also use a `try/finally` block to ensure that the lock is released, even if an exception is thrown.

Listing 9.38—Methods Synchronized Using a Lock

```
class BankAccount {

  private int amt;

  private static Lock lock = new ReentrantLock();

  public BankAccount(int amt) {
    this.amt = amt;
  }

  /**
   * Transfers money from one account to another.
   * @param b2 other bank account to deposit money into.
   * @param amt the amount to withdraw from this account.
   */
  public void transfer(BankAccount b2, int amt) {
    BankAccount.lock.lock();
    try {
      this.withdraw(amt);
      b2.deposit(amt);
    } finally {
      BankAccount.lock.unlock();
    }
  }

  public void deposit(int more) {
    BankAccount.lock.lock();
    try { this.amt += more; }
    finally { BankAccount.lock.unlock(); }
  }

  public void withdraw(int less) {
    BankAccount.lock.lock();
```

```
      try {
        if (amt >= less) { this.amt -= less; }
        else { System.err.printf("withdraw: insufficient funds\n"); }
      } finally {
        BankAccount.lock.unlock();
      }
    }
  }
```

Example 9.12. A *thread-safe data structure* is one that supports reentrancy, i.e., multiple threads working over it at the same time. Java's `ArrayList` class is not thread-safe, so we will implement our own thread-safe array list using locks. The idea is to wrap all pieces of the code that multiple threads may access with a lock. Recalling our `MiniArrayList` class from Chapter **??**, we see that `add`, `remove`, `insert`, `get`, and `size` all either access or modify the state of the list, which could conflict with another thread and cause a dangerous race condition. Thus we will surround the code within a lock. Note that some methods do not, themselves, need to be synchronized, e.g., `shiftLeft/shiftRight/resize`, because they are only ever called from within a synchronized context. Should we wrap those methods in the same lock, then the thread who owns the lock would be unable to call those methods, causing a deadlock.[1] Moreover, because the `remove` method calls `get`, acquiring the lock immediately in `remove` is a bad idea. So, we acquire the lock in `get`, then release it, then acquire it again in `remove`. Certain methods, e.g., `size` and `get`, need to be updated to account for releasing the lock before the return statement; in these cases we store the result in a (local) temporary variable, release the lock, then return. In the code segment, we omit the comments for the sake of brevity. In all cases, we use `try/finally` blocks to ensure that, even in the case of exceptions, the lock is released, which is especially important with data structural methods.

Listing 9.39—Thread-Safe Array List Implementation

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class ThreadSafeArrayList<T> {

  // ...  other instance variables not shown.
  private Lock lock;

  public ThreadSafeArrayList(int capacity) {
    this.size = 0;
    this.capacity = capacity;
    this.elements = (T[]) new Object[capacity];
    this.lock = new ReentrantLock();
  }

  public void add(T element) {
    this.lock.lock();
    if (this.size == this.capacity) { this.resize(); }
    this.elements[this.size] = element;
    this.size++;
    this.lock.unlock();
  }
```

[1] As we showed with the bank account example, Java's `ReentrantLock` class is reentrant, meaning that a thread can safely acquire the lock multiple times. So, this problem is not necessarily an issue, but we will still avoid it for the sake of clarity and adaptability to other languages that may not have reentrant locks.

```java
  public void insert(T e, int idx) {
    this.lock.lock();
    try {
      if (this.size == capacity) { this.resize(); }
      this.shiftRight(idx);
      this.elements[idx] = e;
    } finally { this.lock.unlock(); }
  }

  public T remove(int idx) {
    this.lock.lock();
    try {
      T e = this.get(idx);
      this.shiftLeft(idx);
      this.size--;
    } finally { this.lock.unlock(); }
    return e;
  }

  public T get(int index) {
    this.lock.lock();
    try { T e = this.elements[index]; }
    finally { this.lock.unlock(); }
    return e;
  }

  public int size() {
    this.lock.lock();
    try { int sz = this.size; }
    finally { this.lock.unlock(); }
    return sz;
  }
}
```

Java, in fact, does have a data structure in the Collections API that supports multithreading and is therefore thread-safe: the `Vector` class serves this purpose. Though, should we not want to use `Vector`, since some Java programmers consider it "deprecated," we can invoke the static `synchronizedList` method on `Collections` to return a synchronized/thread-safe version of the input list. There are also other data structures in the Collections API that are thread-safe, such as `ConcurrentHashMap` and `ConcurrentLinkedQueue`, as well as methods to convert a non-thread-safe data structure into one that is thread-safe, e.g., `synchronizedMap` and `synchronizedSet`.

Example 9.13. In the event that we want to block concurrent access to a variable, we need to enforce a few more precautions than simply marking methods that access the variable as `synchronized`. Moreover, should we want to not deal with locks but still require atomic operations over access to an integer, we can use, for example, `AtomicInteger`. Let's revisit the `counter` example from earlier, but this time we will use an `AtomicInteger` to ensure that the increment operation is atomic.

Listing 9.40—Example using `AtomicInteger`

```java
import java.util.concurrent.atomic.AtomicInteger;
import java.lang.Thread;
import java.lang.Runnable;

class AtomicIntegerExample {
```

```java
  private static AtomicInteger counter = new AtomicInteger(0);

  public static void main(String[] args) {
    Thread t1 = new Thread(new Incrementer());
    Thread t2 = new Thread(new Incrementer());

    t1.start();
    t2.start();

    try {
      t1.join();
      t2.join();
    } catch (InterruptedException ex) { ex.printStackTrace(); }

    System.out.println(counter);
  }

  private static class Incrementer implements Runnable {

    @Override
    public void run() {
      for (int i = 0; i < 1_000_000; i++) { inc(); }
    }

    public static synchronized void inc() {
      counter.incrementAndGet();
    }
  }
}
```

Condition Variables

Example 9.14. Imagine that we are writing a multithreaded program, where thread t_1 is a "consumer," and there are multiple "producer" threads $t_2, \ldots, t_n$. The producer threads add data to a list, and the consumer polls them when available. We want to ensure that the consumer only polls the list when it is non-empty. So, one might think to write the following code for the consumer:

Listing 9.41—Consumer Thread Behavior

```java
import java.lang.Runnable;
import java.util.Vector;

class Consumer<T> implements Runnable {

  private final Vector<T> LIST;

  public Consumer(Vector<T> ls) { this.LIST = ls; }

  @Override
  public void run() {
    while (true) {
      while (this.LIST.isEmpty()) { /* Do nothing. */ }
      T e = this.LIST.remove(0);
      System.out.printf("Consumed: %s\n", e.toString());
    }
  }
}
```

The consumer thread will "busy-wait" until the list is non-empty. A busy-wait loop is not recommended because the thread has to continuously check the list to determine if an element exists, which is wasteful. Plus, there's the added problem that this code is not thread-safe; by not acquiring a lock, we run into the potential of a race condition where one thread polls the queue and another is about to poll the queue. Even though our queue definition *is* thread-safe, this does not solve the problem of a thread trying to remove an element that does not exist outside the fact. The solution is to use a *condition variable*, which serves as a signal between threads. While the list is empty, our consumer thread awaits on the condition variable, thereby putting it to sleep. Then, the producer thread(s) will issue a signal to the condition variable when inserting data into the list. Condition variables are always associated with a lock, so we need to define one in our main program and pass it around to both the producer(s) and consumer.

When the consumer acquires the lock, it checks if the list is empty. If so, it awaits on the condition variable, which releases the held lock and puts the thread to sleep. When the producer acquires the lock, it adds an element to the list, then issues a signal to the condition variable, which wakes up the consumer thread. The signal causes the consumer to reacquire the lock, check if the list is empty, and, if not, remove (and process) the head element from the list, followed by releasing the lock.

For the sake of our example, suppose that a producer adds data to the list using a random number generator.[1] That is, we generate a random number, determine if it is within a specific range and, if so, add data to the list. This then means that the producer thread signals on the condition variable, awaking the consumer.

Listing 9.42—Producer Thread Behavior

```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.Random;
import java.util.Vector;

class Producer implements Runnable {

  private static final Random RAND = new Random();

  private final Vector<Integer> LIST;
  private final Lock LOCK;
  private final Condition COND_VAR;

  public Producer(Vector<Integer> ls, Lock lock, Condition condVar) {
    this.LIST = ls;
    this.LOCK = lock;
    this.COND_VAR = condVar;
  }

  @Override
  public void run() {
    while (true) {
      this.LOCK.lock();
      int n = this.RAND.nextInt(100);
      if (n < 10) {
        this.LIST.add(n);
        this.COND_VAR.signal();
```

[1]The Random class is thread-safe.

```
        }
        this.LOCK.unlock();
      }
    }
  }
```

Further note that awaiting on a condition variable may throw an `InterruptedException`, meaning we must surround the call with a `try/catch` block.[1] An important fact to also consider is that, if a thread that issues a signal on a condition variable does not own the respective lock, Java will throw an `IllegalMonitorStateException`.

Listing 9.43—Producer/Consumer Thread Behavior

```java
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.Random;
import java.util.Vector;
import java.lang.Thread;

class ConditionVariableExample {

  private static final Lock LOCK = new ReentrantLock();
  private static final Condition COND_VAR = LOCK.newCondition();
  private static final Vector<Integer> LIST = new Vector<>();

  public static void main(String[] args) {
    Thread t1 = new Thread(new Consumer<>(LIST, LOCK, COND_VAR));
    Thread t2 = new Thread(new Producer<>(LIST, LOCK, COND_VAR));
    Thread t3 = new Thread(new Producer<>(LIST, LOCK, COND_VAR));

    t1.start();
    t2.start();
    t3.start();
  }
}
```

If we fix our input seed to `212`, the output is deterministically as follows:

```
Consumed: 1
Consumed: 6
Consumed: 1
Consumed: 7
Consumed: 7
Consumed: 8
Consumed: 1
Consumed: 4
Consumed: 5
```

Networking & Sockets

Computers that communicate with one another is not a revolutionary idea, but being able to write programs that act as *clients* and *servers* is certainly cool. A server is a program that listens for incoming connections over a network and processes them accordingly. Clients connect to servers to do different things, e.g., receive information from a server,

[1] Should we not want to do this, we can instead invoke the `awaitUninterruptibly` method.

talk to other connected clients, and so on. In this section we will demonstrate the Java networking/socket API and how to write a few programs that incorporate a server.

Example 9.15. The simplest kind of server is one that receives an incoming connection and outputs/relays some data to the client, then closes said connection. Our first example using a server will accept a client, then tell it the current server clock date and time, then close the connection. Servers operate on *ports* over a connection, which, simply put, are virtual places where connections occur. Our server will be hosted on port 8080: a common default port. So, we instantiate an object of type `ServerSocket` to listen on 8080, and set up an infinite loop to forever listen for connections. Upon receiving one, we accept the client as a `Socket`, we instantiate a data stream to print information out to the client. One important detail to note is that we need to designate the `PrintWriter` output stream to "autoflush" its data. In essence, this means that as soon as the client reads data from the server, it gets emitted to their standard output stream. Note that we do not need to declare a reader stream from the client, since our server does not receive data.

The server *blocks* until it receives a connection, similar to how many of the input reader classes, e.g., `Scanner`, `BufferedReader`, wait until data is sent to the input stream to continue execution. A blocking server that is implemented in this fashion, as we have done, can only serve one client at a time, since it has to accept the client, set up the output stream, print the data, then close the connection before being able to accept another.

Listing 9.44—Time Echo Server

```java
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;

class TimeEchoServer {

  private static final int PORT = 8080;

  public static void main(String[] args) {
    try (ServerSocket ss = new ServerSocket(PORT)) {
      System.out.printf("Server listening on port %d...\n", PORT);
      // Continuously listen for clients.
      while (true) {
        // Accept the incoming client, block until we receive one.
        Socket client = ss.accept();
        System.out.printf("Client connected: %s\n",
                          client.getInetAddress().toString());
        PrintWriter pw = new PrintWriter(
                          new OutputStreamWriter(client.getOutputStream()),
                                                 true);
        // Echo the server time to the client.
        pw.println("The server time is " + java.time.LocalDateTime.now());
        // Echo that the client has disconnected, then close their connection.
        System.out.printf("Client disconnected: %s\n",
                          client.getInetAddress().toString());
        client.close();
      }
    } catch (IOException ex) { ex.printStackTrace(); }
  }
}
```

To connect to this server, we can use the Unix `nc` "netcat" command to connect to `localhost`, e.g., `nc localhost 8080`, while the server is running.

Example 9.16. Let's write a server that, upon receiving a connection, allows the client to type a number. This number is then translated by the server into an index, grabbing them the n^{th} most populous country. So, our server first must read in a list of countries from most populous to least. From there we establish the same server connection, only this time we must use both the input and output streams of the client, so we can read the number that they enter, as well as output the corresponding country.

Listing 9.45—Country Rank Server

```java
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.List;

class CountryPopulationRankServer {

  private final int PORT;
  private final List<String> COUNTRIES;

  public CountryPopulationRankServer(int port) {
    this.PORT = port;
    this.COUNTRIES = new ArrayList<>();

    // Read in the words from the text file.
    try (BufferedReader br = new BufferedReader(new FileReader("clist.txt"))) {
      String line = null;
      while ((line = br.readLine()) != null) { this.COUNTRIES.add(line); }
    } catch (IOException ex) { ex.printStackTrace(); }
  }

  public void start() {
    // Set up the thread pool.
    try (ServerSocket ss = new ServerSocket(this.PORT)) {
      while (true) {
        Socket skt = ss.accept();
        BufferedReader br = new BufferedReader(
                              new InputStreamReader(skt.getInputStream()));
        // Read in the country from the user.
        String line = br.readLine();
        int country = Integer.parseInt(line) - 1;
        PrintWriter pw = new PrintWriter(socket.getOutputStream(), true);
        // Output the nth most populous country.
        pw.printf("%d country: %s\n", country + 1, this.COUNTRIES.get(country));
        socket.close();
      }
    } catch (IOException ex) { ex.printStackTrace(); }
  }

  public static void main(String[] args) {
    new CountryPopulationRankServer(8080).start();
  }
}
```

Example 9.17. Let's use what we have learned to write a *multithreaded* chat server. That is, we will write a program that acts as server for a chat room. Clients can connect to the server and send messages to each other. The server will be multithreaded so that it can simultaneously serve clients. First, we need to decide on a protocol for communication between the server and the clients. For this example, to minimize the number of features necessary to convey the important ideas, we will only allow the user to log into the "room," type messages to all users in the room, or to disconnect via the `\quit` command. The server houses a *thread pool* that spins up a thread to communicate with a connected client. Fortunately for us, because we are in Java, we have access to synchronized data structures such as `Vector`. Therefore, we do not need to manually insert locks into a data structure for storing those connected clients. After a server receives a client, as we stated, we dedicate a thread to that user, then resume listening. We will also have a server thread that actively listens for messages that are then relayed to every other user. The notion of the server having a thread to listen for messages is known as a *task-handler*, whose tasks are queued away in a synchronized `LinkedBlockingQueue` data structure. When reading a command from the user, we enqueue it into the server's queue of tasks, which blocks until a task is available. The task-handler then dequeues the task and broadcasts it to all users. It will be up to the client to determine whether or not they are the intended recipient. [1]

Listing 9.46—Task Handler Class Definition

```java
import java.lang.Runnable;

class Server {
  // ...  other data not shown.

  private static class TaskHandler implements Runnable {

    private final Server SERVER;

    public TaskHandler(Server server) { this.SERVER = server; }

    @Override
    public void run() {
      while (this.SERVER.running) {
        try {
          Task task = this.SERVER.TASKS.take();
          this.SERVER.CLIENTS.forEach(c -> c.send(task));
        } catch (InterruptedException e) { e.printStackTrace(); }
      }
    }
  }
}
```

Listing 9.47—Server Implementation

```java
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.Vector;
import java.net.ServerSocket;
import java.net.Socket;
```

[1] From a security standpoint, broadcasting a message in this fashion is incredibly insecure, since a client can simply decide if it is the intended recipient no matter if the message is not intended for them.

```java
import java.io.IOException;

class Server {

  private final ExecutorService THREAD_POOL;
  private final Vector <Client> CLIENTS;
  private final BlockingQueue <Task> TASKS;
  private final int PORT;
  private boolean running;

  public Server(int port) {
    this.running = true;
    this.PORT = port;
    this.THREAD_POOL = Executors.newCachedThreadPool();
    this.CLIENTS = new Vector<>();
    this.TASKS = new LinkedBlockingQueue<>();
    this.THREAD_POOL.execute(new TaskHandler(this));
  }

  public void start() {
    try {
      ServerSocket server = new ServerSocket(this.PORT);
      System.out.printf("Server connected on port %d...\n", this.PORT);
      while (this.running) {
        Socket client = server.accept();
        System.out.printf("Client %s connected!\n", client.getInetAddress());
        Client c = new Client(client, this);
        this.CLIENTS.add(c);
        this.THREAD_POOL.execute(c);
      }
    } catch (IOException e) { e.printStackTrace(); }
  }

  public void addTask(Task t) {
    if (t != null) { this.TASKS.add(t); }
  }

  public static void main(String[] args) { new Server(8080).start(); }
}
```

Clients contain an output and input stream, which are used to send and receive messages to and from the server, as we saw with the country population ranking server. The client thread is responsible for reading messages from the server and printing them to the user's console, as well as receiving messages from the clients' standard input stream, and feeding those lines to the server. We will also designate that a client has a user identifier and stores a flag indicating whether or not the client is "logged in", which simply represents if they can receive broadcast messages. By default, the user identifier will be the IP address of the connected socket, which is replaceable/alterable via the `"login"` command. Moreover, in case a client disconnects from the server, we utilize the `connected` flag to prevent the thread from reusing the now-closed input stream. We will disable this flag once we implement the functionality for handling quit commands.

Listing 9.48—Client Implementation

```java
import java.io.IOException;
import java.io.BufferedReader;
import java.io.PrintWriter;
import java.lang.Runnable;
```

```java
import java.net.ServerSocket;
import java.net.Socket;

class Client implements Runnable {

  private final Socket SOCKET;
  private final Server SERVER;
  private final BufferedReader IN;
  private final PrintWriter OUT;

  private boolean loggedIn;
  private boolean connected;
  private String userId;

  public Client(Socket socket) {
    this.SOCKET = socket;
    this.SERVER = server;
    this.IN = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    this.OUT = new PrintWriter(socket.getOutputStream(), true);
    this.loggedIn = false;
    this.connected = true;
    this.userId = socket.getInetAddress().toString();
  }

  @Override
  public void run() {
    try {
      String line;
      while (this.connected && (line = this.IN.readLine()) != null) {
        this.SERVER.addTask(parseCmd(line));
      }
    } catch (IOException e) { e.printStackTrace(); }
  }
}
```

We now need to write the `parseCmd` message, which receives an unparsed string from the client and returns a `Task` object, which we will design momentarily. Commands, in our system, will be prefixed via a forward slash /, followed by the arguments thereof. As an example, `/login joshua` will set the user's identifier to `joshua`. We will also allow the user to send messages to all other users via the `/send` command, which is followed by the message broadcast. Finally, the `/quit` command disconnects the user from the server.

Listing 9.49—Parsing the Input Command

```java
class Client {
  // ...  other information not shown.

  public Task parseCmd(String line) {
    String cmd = line.substring(0, line.indexOf(' '));
    String msg = line.substring(line.indexOf(' ') + 1);
    switch (cmd) {
      case "/login" -> { return this.handleLoginCmd(msg); }
      case "/send" -> { return this.handleSendCmd(msg); }
      case "/quit" -> { return this.handleQuitCmd(); }
      default -> throw new IllegalArgumentException("parseCmd: bad cmd " + cmd);
    }
  }
}
```

Our tasks will consist of a hierarchy of objects. Namely, a `Task` contains an enumeration of its type, as well as the raw data. A `SenderTask` is a task that contains a sender information, which stores an instance variable denoting the `Client` who the message originated from. Finally, a `BroadcastTask` is a task that contains a message to be broadcast to all users, which is itself a subclass of `SenderTask`. Tasks are operations that must be sent to the server and handled by one of its threads. Certain commands may remain local to the server due to their simplicity, e.g., `"/login"` and `"/quit"` do not need to be relayed to other users (and therefore are not wrapped as `Task` instances).

Listing 9.50—Task Type Enumeration

```java
enum TaskType {
  BROADCAST
}
```

Listing 9.51—Task Class

```java
abstract class Task {

  private final TaskType TYPE;
  private final String RESPONSE;

  public Task(TaskType type, String response) {
    this.TYPE = type;
    this.RESPONSE = response;
  }

  public TaskType getType() { return this.TYPE; }

  public String getResponse() { return this.RESPONSE; }
}
```

Listing 9.52—SenderTask Class

```java
abstract class SenderTask extends Task {

  private final Client SENDER;

  public SenderTask(TaskType type, String response, Client sender) {
    super(type, response);
    this.SENDER = sender;
  }

  public Client getSender() { return this.SENDER; }
}
```

Listing 9.53—BroadcastTask Class

```java
class BroadcastTask extends SenderTask {

  public BroadcastTask(Client sender, String response) {
    super(TaskType.BROADCAST, response, sender);
  }
}
```

We now must write the `Client` `send` method, which receives a `Task` from the server and interprets it accordingly. The only task that our server can send is `BROADCAST`, so we simply need to check whether the client is logged in and, if so, print the message to their output stream. At the same time, we can also take care of privatized parsing command methods. Only one of these three should return a non-`null` object. It is also a design decision to make `handleQuitCmd` and `handleLoginCmd` return `null` because they do not need to be broadcast to other users. We do so to ensure consistency among the methods.

Listing 9.54—Client `send` Method

```java
class Client {
  // ...  other information not shown.

  public void send(Task t) {
    switch (t) {
      case BroadcastTask tb -> {
        if (this.loggedIn) {
          this.OUT.printf("%s: %s\n", tb.getSender(), tb.getResponse());
        }
      }
      default -> {
        throw new IllegalArgumentException("send: bad task " + t.getType());
      }
    }
  }

  private Task handleLoginCmd(String msg) {
    this.userId = msg;
    this.loggedIn = true;
    return null;
  }

  private Task handleSendCmd(String msg) {
    return new BroadcastTask(this, msg);
  }

  private Task handleQuitCmd() {
    this.loggedIn = false;
    try {
      this.SOCKET.close();
      this.IN.close();
      this.OUT.close();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      return null;
    }
  }
}
```

Now we can run the server! Clients connect via the "netcat" command `nc`, which is a Unix utility for reading and writing to network connections. We can connect to the server, using multiple terminals, via `nc localhost 8080`. Below is a demo of the server running with three clients connected.

9.5 Design Patterns

Part of the reason that we have spent so much time discussing object-oriented design is because it is a fundamental aspect of software engineering. In particular, we have discussed the importance of designing classes that are cohesive, loosely coupled, and adhere to the single responsibility principle. We have also discussed the importance of designing classes that are extensible, and how to do so through the use of inheritance and polymorphism. As software engineers, we are responsible for designing and implementing software that is scalable and maintainable. Scalable software does not explode in complexity when adding new features/functionality, or as the user-base grows. Maintainable software, on the other hand, is simple to understand and make incremental changes and fixes as time passes.

In this section we will discuss several *design patterns*, which are common solutions to specific problems that arise largely in the context of object-oriented design. Design patterns are, of course, not specific to Java and can be implemented in any reasonable object-oriented language.

Command

The *command* pattern is a simple pattern that encapsulates a request, of sorts, to some type of handler. The handler knows nothing about the request itself, only that the handler acts as a dispatch for invoking the request.

Example 9.18. Suppose that we're writing a game that involves moving a player around an environment. We want to write a class that handles moving the player, but remains independent of the player implementation. First, we will say that the player executes some type of `Command`, which is an interface containing a sole `execute` method. Then, we will write a `Player` class containing `move` and `jump` methods, where the former increments their x coordinate and jump increments their y coordinate, starting from the origin.

Listing 9.55—Command Interface Definition

```
interface Command {

  void execute();
}
```

Listing 9.56—Player Class Definition

```
class Player {

  private int x;
  private int y;

  public Player() { this.x = 0; this.y = 0; }

  // Getters and setters omitted.
}
```

Now, we will write two subtypes of `Command`, namely `MoveCommand` and `JumpCommand`, that each implement `execute`, the only difference being the intended behavior. The

MoveForward command receives the Player instance and a direction, whereas JumpCommand only needs to receives a Player instance.

Listing 9.57—Move Command Implementation

```
class MoveCommand implements Command {

  private Player player;

  public MoveCommand(Player p) { this.player = p; }

  @Override
  public void execute() { this.player.setX(this.player.getX() + 1); }
}
```

Listing 9.58—Jump Command Implementation

```
class JumpCommand implements Command {

  private Player player;

  public JumpCommand(Player p) { this.player = p; }

  @Override
  public void execute() { this.player.setY(this.player.getY() + 1); }
}
```

Lastly, we must implement a "command dispatch handler," of sorts, which we might envision to be a controller. In particular, we will write InputHandler to store two commands that respond to presses to an 'X button' and presses to a 'Y button'. The methods to 'press' each button correspond to invoking execute on the respective commands. The general idea behind the command pattern is that we can pass any arbitrary implementation of a command to this handler to change/update the behavior of a button press or action in the handler.

Listing 9.59—Input Handler/Command Controller

```
class InputHandler {

  private Command xButton;
  private Command yButton;

  public InputHandler(Command xButton, Command yButton) {
    this.xButton = xButton;
    this.yButton = yButton;
  }

  public void pressXButton() { this.xButton.execute(); }

  public void pressYButton() { this.yButton.execute(); }
}
```

Now, we can test the system. Under other circumstances we would write JUnit tests before writing the separate commands, but we needed to write the handler before writing coherent tests.

Listing 9.60—Command Tester

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

class CommandTester {

  @Test
  void testCommand() {
    Player p = new Player();
    InputHandler handler = new InputHandler(new MoveCommand(p),
                                            new JumpCommand(p));
    assertAll(
      () -> assertEquals(0, p.getX()),
      () -> assertEquals(0, p.getY()),
      () -> handler.pressXButton(),
      () -> assertEquals(1, p.getX()),
      () -> handler.pressYButton(),
      () -> assertEquals(1, p.getY()));
  }
}
```

As shown, we have decoupled the player from the handler, and the handler from the implementation of the commands. This, consequently, allows us to alter or modify the behavior of commands without redesigning the handler or player classes.

Factory

Example 9.19. To showcase our next pattern, we will design the `Fraction` class to represent mathematical fractions containing integer numerators and denominators. This example greatly resembles the rational number class in a previous chapter, but we introduce a twist to exemplify the benefits of the *factory* pattern.

Listing 9.61

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

class FractionTester {

  @Test
  void testFraction() {
    assertAll(
      () -> assertEquals("1/1", new Fraction(1, 1).toString()),
      () -> assertEquals("1/2", new Fraction(1, 2).toString()),
      () -> assertEquals("17/312", new Fraction(17, 312).toString()),
      () -> assertEquals("321/199", new Fraction(321, 199).toString()));

    // Test random allocations of 1/2.
    for (int i = 0; i < 500_000; i++) {
      assertEquals("1/2", new Fraction(1, i).toString());
    }
  }
}
```

Listing 9.62

```
class Fraction {

  private int num;
  private int den;

  public Fraction(int num, int den) {
    this.num = num;
    this.den = den;
  }

  @Override
  public String toString() { return String.format("%d/%d", this.num, this.den); }
}
```

Notice that one of our test cases loops five hundred thousand times, repeatedly allocating the same fraction, namely 1/2. It is almost certainly true that, whatever application uses the `Fraction` class, will not need separate/distinct instances of a fraction. Accordingly, we are unnecessarily allocating `Fraction` instances, taking a lot of CPU time and memory. The solution is to introduce a form of caching, wherein we create a lookup table of the most "common" fractions and, whenever someone wants to construct a `Fraction`, we first determine if it can be polled from the table. If not, we have no choice but to allocate the fraction. We call this the *factory* pattern, because we have a class that represents and processes the creation of `Fraction` objects, rather than allowing the user to directly instantiate one themselves.

In designing the `FractionFactory` class, we declare a five-hundred element array to store the fractions $1/n$, where n is an integer such that $1 \leq n \leq 500$. Its constructor allocates the fraction "cache." We now need a method to take the role of building fractions; namely a `Fraction create(int num, int den)` method, which either looks up and returns the shared instance of a common fraction, or allocates a new instance.

Listing 9.63

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

class FractionFactoryTester {

  @Test
  void testFractionFactory() {
    FractionFactory ff = new FractionFactory();
    assertAll(
      () -> assertEquals("1/1", ff.create(1, 1).toString()),
      () -> assertEquals("1/2", ff.create(1, 2).toString()),
      () -> assertEquals("17/312", ff.create(17, 312).toString()),
      () -> assertEquals("321/199", ff.create(321, 199).toString()));

    // Check to see that all of the allocations are the same.
    Fraction f1 = ff.create(1, 2);
    for (int i = 0; i < 500_000; i++) {
      assertEquals("1/2", ff.create(1, 2).toString());
      assertEquals(true, f1 == ff.create(1, 2));
    }
  }
}
```

Listing 9.64

```
class FractionFactory {

  private static final int LIMIT = 500;

  private static final Fraction[] CACHE;

  public FractionFactory() {
    this.CACHE = new Fraction[LIMIT];
  }

  /**
   * Creates a new Fraction instance, or looks it up in the cache.
   * @param num numerator of fraction.
   * @param den denominator of fraction.
   * @return a new instance of Fraction, or a shared instance if it was cached.
   */
  public Fraction create(int num, int den) {
    return den >= 1 && den <= LIMIT ? this.CACHE[den - 1]
                                    : new Fraction(num, den);
  }
}
```

Example 9.20. Imagine that we have a file of data containing information about a "Person," which entails their university records. A person can be a student, faculty, or staff member, and each type of person contains a name. For the purposes of this example, this is the only information that we care about, but we could easily extend it to include other datapoints. The idea is that we want to read in these records and create a `Person` object for each record. We could store 'type' of person as, say, an enum, or a string, but this is not ideal. Instead, we will use the *factory* pattern to create a `Person` object for each record, and the factory will determine the subclass of person based on the record.

Namely, let's create the `abstract Person` class, which contains a `name` field, a `getName` method, and an `abstract public String getRole()` method to-be overridden in each subclass. We will also write the `Student`, `Faculty`, and `Staff` classes, which extend `Person` and contain the overridden `getRole` method that returns the type of person. Instead of directly instantiating `Person` objects, we take advantage of the factory pattern by writing the `PersonFactory` class, which contains a `create` method that takes a `name` and `role` as arguments, and returns the relevant subclass of `Person`.

To read the input file, we will write the `PersonDatabase` class, which stores a `Map` of `Person` objects, where the key is the name of the person and the value is the `Person` object. The `PersonDatabase` class contains a `readFile` method that takes a `String` argument representing the path to the file, and reads the file line-by-line, creating a `Person` object for each record and storing it in the `Map`.

Listing 9.65—Reads a File of Person Records

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

class PersonDatabase {
```

```java
  private final Map<String, Person> DATABASE;

  public PersonDatabase() { this.DATABASE = new HashMap<>(); }

  public void readFile(String path) {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
      String line;
      while ((line = br.readLine()) != null) {
        String[] tokens = line.split(",");
        this.DATABASE.put(tokens[0], PersonFactory.create(tokens[0], tokens[1]));
      }
    } catch (IOException ex) { ex.printStackTrace(); }
  }

  public Person lookup(String name) { return this.DATABASE.get(name); }
}
```

Listing 9.66—Person Factory Implementation

```java
class PersonFactory {

  /**
   * Creates a new Person object based on the role.
   * @param name name of person.
   * @param role role of person.
   * @return a new subclass of Person.
   */
  public static Person create(String name, String role) {
    return switch(role) {
      case "student" -> new Student(name);
      case "faculty" -> new Faculty(name);
      case "staff" -> new Staff(name);
      default -> throw new IllegalArgumentException("create: invalid role");
    };
  }
}
```

Listing 9.67—Abstract Person Class Definition

```java
abstract class Person {

  private final String NAME;

  public Person(String name) { this.NAME = name; }

  public String getName() { return this.NAME; }

  public abstract String getRole();
}
```

Listing 9.68—Student Class Definition

```java
class Student extends Person {

  public Student(String name) { super(name); }

  @Override
  public String getRole() { return "student"; }
}
```

Listing 9.69—Faculty Class Definition

```
class Faculty extends Person {

  public Faculty(String name) { super(name); }

  @Override
  public String getRole() { return "faculty"; }
}
```

Listing 9.70

```
class Staff extends Person {

  public Staff(String name) { super(name); }

  @Override
  public String getRole() { return "staff"; }
}
```

If we assume that our input records are line-separated and comma-delimited, then we can write a simple test case to verify that the `PersonDatabase` class correctly reads the file and creates the appropriate `Person` objects. The following file (titled `"records1.dat"`) will be used for testing purposes.

```
Willard Van Orman Quine,faculty
Alan Mathison Turing,student
John von Neumann,staff
Stephen Cole Kleene,faculty
```

Listing 9.71—Person Database Tester

```
import static Assertions.assertAll;
import static Assertions.assertTrue;

class PersonDatabaseTester {

  @Test
  void testPersonDatabase() {
    PersonDatabase db = new PersonDatabase();
    db.readFile("records1.dat");
    assertAll(
      () -> assertTrue(db.lookup("Willard Van Orman Quine") instanceof Faculty),
      () -> assertTrue(db.lookup("Alan Mathison Turing") instanceof Student),
      () -> assertTrue(db.lookup("John von Neumann") instanceof Staff),
      () -> assertTrue(db.lookup("Stephen Cole Kleene") instanceof Faculty));
  }
}
```

As expected, all tests pass.

Builder

When creating instances of classes, it is not always feasible or possible to pass all of the necessary arguments to a constructor. The *builder* pattern allows us to write "partial constructors," i.e., methods that take a subset of the arguments and return an object that contains the partial arguments.

Example 9.21. Suppose that we are designing a `URL` class to represent a URL, which contains a schema, host, port, and path. We want to write a `URLBuilder` class that allows us to construct a `URL` object by passing arguments one-at-a-time to a series of methods. In particular, each instance variable, namely `_schema`, `_host`, `_port`, and `_path`, has a corresponding method of the same name sans the underscore. Each method returns `this`, which allows us to chain method calls together. Moreover, returning `this` and forgoing the constructor means we do not need to unnecessarily allocate a new `Url` object for every method call.

We will designate that a "complete" URL is one that has, at the very least, a schema and a host. To complement this, we now design the `build` method that returns a `Url` object if the schema and host are non-`null`, otherwise, it throws an `IllegalStateException`. When the optional fields are not specified, they are set to default values, namely `0` for the port and `""` for the path.

To test the implementation, we chain together a series of method calls on a `Url` instance and verify that the resulting `Url` object is correct through its `toString` representation.

Listing 9.72—Url Tester

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

class UrlTester {

  @Test
  void testUrlBuilder() {
    Url url = new Url().schema("http")
                       .host("www.google.com")
                       .port(80)
                       .path("search");
    assertAll(
      () -> assertEquals("http://www.google.com:80/search", url.toString()));
  }
}
```

Listing 9.73—Url Class Definition

```
class Url {

  private String _schema;
  private String _host;
  private int _port;
  private String _path;

  private Url() {}

  public Url schema(String schema) {
    this._schema = schema;
    return this;
  }

  public Url host(String host) {
    this._host = host;
    return this;
  }

  public Url port(int port) {
```

```java
    this._port = port;
    return this;
  }

  public Url path(String path) {
    this._path = path;
    return this;
  }

  public Url build() {
    if (this._schema == null || this._host == null) {
      throw new IllegalStateException("build: incomplete URL");
    }
    this._port = this._port == 0 ? 80 : this._port;
    this._path = this._path == null ? "" : this._path;
    this.complete = true;
    return this;
  }

  @Override
  public String toString() {
    return String.format("%s://%s:%d/%s", this._schema,
                         this._host, this._port, this._path);
  }
}
```

Partially-constructed objects may seem odd at first, but they are useful in situations where we want to instantiate an object piecemeal, i.e., one instance variable at a time. Plus, we can reuse the same builder with multiple objects. As an example, suppose that we want to create a `Url` for a particular host and schema, but without a specific port or path. We can then reuse this object repeatedly to populate an partially-constructed instance with differing ports and paths, rather than having to unnecessarily repeat the known schema and host.

Visitor

The *visitor* pattern is the most complex pattern that we will work with, but it offers a host of benefits. Consider a situation in which we do not have access to classes, but want to modify their implementation to add some functionality. In particular, if those classes support the use of visitors, then we can design almost any type of functionality without needing to modify those classes at all.

Example 9.22. Let's write a visitor that prints out a programming language expression in a human-readable format. In particular, we will use a simplified version of the interpreter from Chapter 5. First, we need to write the visitor interface, which contains a `visit` method for each type of expression, namely `NumNode`, `PrimNode`, `VarNode`, and `LetNode`. Each `visit` method takes an expression of the corresponding type as an argument and returns a string.

Listing 9.74—The `ExpressionVisitor` Interface

```java
interface ExpressionVisitor {

  String visit(NumNode node);
  String visit(PrimNode node);
  String visit(VarNode node);
  String visit(LetNode node);
```

```
}
```

Next, we need to modify the AstNode class to include an abstract visit method that receives a Visitor object and calls the appropriate visit method using polymorphic dispatch.

Listing 9.75—The AstNode Class

```
class AstNode {
  // ...  other methods and variables not shown.

  abstract String visit(ExpressionVisitor visitor);
}
```

Now, we update each subclass to override the visit method and call the appropriate visit method. Fortunately this is trivial, since all we must do is add the method signature and call visit with this as the argument to represent that the visitor is visiting the current node. Because this is consistently redundant, we will only show the implementation of the NumNode class, but the remaining classes are identical with respect to this method.[1]

Listing 9.76—The NumNode Class

```
class NumNode extends AstNode {
  // ...  other methods and variables not shown.

  public String visit(ExpressionVisitor visitor) {
    return visitor.visit(this);
  }
}
```

From here, we need to design a variant of the interface that implements the expression printing behavior. Thus we will write the ExpressionPrinterVisitor class, which implements the ExpressionVisitor interface. The ExpressionPrinterVisitor class overrides the respective methods from the ExpressionVisitor interface to prints out the expression, to standard output, in a "stringified" format.

The corresponding tester is nothing different from previous tests; we instantiate an instance of ExpressionVisitor to ExpressionPrinterVisitor, followed by a call to visit on the root node of the expression tree. The result is a string representation of the expression, which we then verify.

Listing 9.77—The ExpressionPrinterVisitorTester Class

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

class ExpressionPrinterVisitorTester {

  @Test
  void testPrimExprPrint() {
    AstNode expr = new PrimNode("+",
                       new NumNode(1),
```

[1]Remember that we said that we could not alter/update these classes (or rather that the visitor pattern guarantees this invariant), but we rely on the assumption that they support the visitor pattern, which is the only internal modification.

```
                        new PrimNode("*",
                          new NumNode(2),
                          new NumNode(3)));
    String result = expr.visit(new ExpressionPrinterVisitor());
    assertAll(
      () -> assertEquals("(+ 1 (* 2 3))", result));
  }

  @Test
  void testLetExprPrint() {
    AstNode expr = new LetNode("x",
                        new NumNode(1),
                        new PrimNode("+",
                          new VarNode("x"),
                          new NumNode(2)));
    String result = expr.visit(new ExpressionPrinterVisitor());
    assertAll(
      () -> assertEquals("(let ([x 1])\n (+ x 2))", result));
  }
}
```

Listing 9.78—The `ExpressionPrinterVisitor` Class

```
import java.lang.StringBuilder;

class ExpressionPrinterVisitor implements ExpressionVisitor {

  @Override
  public String visit(NumNode node) {
    return String.valueOf(node.getValue());
  }

  @Override
  public String visit(PrimNode node) {
    StringBuilder sb = new StringBuilder();
    sb.append("(");
    sb.append(node.getOperator() + " ");
    sb.append(node.getChildren().stream()
                                .map(c -> c.visit(this))
                                .collect(Collectors.joining(" ")));
    sb.append(")");
    return sb.toString();
  }

  @Override
  public String visit(VarNode node) {
    return node.getValue();
  }

  @Override
  public String visit(LetNode node) {
    StringBuilder sb = new StringBuilder();
    sb.append("(let ([" + node.getVar() + " ");
    sb.append(node.value.visit(this));
    sb.append(")]\n ");
    sb.append(node.body.visit(this));
    sb.append(")");
    return sb.toString();
  }

  @Override
```

```
  public String visit(IfNode node) {
    StringBuilder sb = new StringBuilder();
    sb.append("(if ");
    sb.append(node.condition.visit(this));
    sb.append(" ");
    sb.append(node.thenExpr.visit(this));
    sb.append(" ");
    sb.append(node.elseExpr.visit(this));
    sb.append(")");
    return sb.toString();
  }
}
```

Example 9.23. As another example, suppose that we have classes to represent items at a grocery store. Namely, we have a `IGroceryItem` interface, and subtypes `Fruit`, `Milk`, and `Cereal`. Additionally, we can extend the `IGroceryItemVisitor` interface, which itself contains corresponding "visit" methods for each subtype to describe how we wish to visit an `IGroceryItem` object. Let's take advantage of Java's generics to allow a specification of the return type for `visit` methods. A visitor that always returns nothing severely limits the capabilities of the visitor pattern.

Listing 9.79

```
interface IGroceryItemVisitor<R> {

  R visit(Fruit fruit);
  R visit(Milk milk);
  R visit(Cereal cereal);
}
```

Listing 9.80

```
interface IGroceryItem {

  <R> R visit(IGroceryItemVisitor<R> visitor);
}
```

The subtypes of `IGroceryItem`, as we stated, are `Fruit`, `Milk`, and `Cereal`. Fruits have a name and a weight in ounces, milk is either skim milk or regular (designated by a boolean) and a weight in fluid ounces, and cereal has a mascot and a weight in ounces. While we are in the realm of modern Java, let's once again use records to our advantage to remove redundant code. Conveniently, this means that we only have to override the `visit` method in each record type.

Listing 9.81

```
record Fruit(String name, int oz) implements IGroceryItem {

  @Override
  public <R> R visit(IGroceryItemVisitor<R> visitor) {
    return visitor.visit(this);
  }
}
```

Listing 9.82

```
record Milk(boolean skim, int fluidOz) implements IGroceryItem {

  @Override
  public <R> R visit(IGroceryItemVisitor<R> visitor) {
    return visitor.visit(this);
  }
}
```

Listing 9.83

```
record Cereal(String mascot, int oz) implements IGroceryItem {

  @Override
  public <R> R visit(IGroceryItemVisitor<R> visitor) {
    return visitor.visit(this);
  }
}
```

Now, let's extend the capabilities of our classes by designing a visitor that calculates the total price of a grocery list. Namely, we will write the `GroceryListTotalVisitor` class, which implements the `IGroceryItemVisitor` interface. The `GroceryListTotalVisitor` class overrides the respective methods from the `IGroceryItemVisitor` interface to calculate the total price of the grocery list. For the sake of example, fruits are priced at $0.25 per ounce, milk is priced at $2.00 by the gallon, and cereal is priced at $0.10 per ounce. If the milk is skim, we add $0.50 to its price.

The corresponding tester is nothing different from previous tests; we instantiate an instance of `IGroceryItemVisitor` to `GroceryListTotalVisitor`, followed by a call to `visit` on each item in the grocery list. The result is the total price of the grocery list, which we then verify.

Listing 9.84

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

class GroceryListTotalVisitorTester {

  @Test
  void testGroceryListTotalVisitor() {
    List<IGroceryItem> groceryList = List.of(
      new Fruit("apple", 4),
      new Fruit("banana", 2),
      new Milk(true, 128),
      new Cereal("Tony the Tiger", 16)
    );

    assertAll(
      () -> assertEquals(0,
                     List.of().stream()
                              .mapToDouble(item ->
                               item.visit(new GroceryListTotalVisitor()))
                              .sum()),
      () -> assertEquals(5.60,
                     groceryList.stream()
                                .mapToDouble(item ->
```

```
                                        item.visit(new GroceryListTotalVisitor()))
                                       .sum()));
  }
}
```

Listing 9.85

```
class GroceryItemPriceVisitor implements IGroceryItemVisitor<Double> {

  private static final double FRUIT_PRICE_PER_OZ = 0.25;
  private static final double MILK_PRICE_PER_GALLON = 2.0;
  private static final double CEREAL_PRICE_PER_OZ = 0.1;

  @Override
  public Double visit(Fruit f) {
    return f.oz() * FRUIT_PRICE_PER_OZ;
  }

  @Override
  public Double visit(Milk m) {
    double gallonsPrice = m.fluidOz() / 128.0 * MILK_PRICE_PER_GALLON;
    return m.skim() ? gallonsPrice + 0.5 : gallonsPrice;
  }

  @Override
  public Double visit(Cereal c) {
    return c.oz() * CEREAL_PRICE_PER_OZ;
  }
}
```

9.6 API Connectivity

Writing programs that interact with the outside world is outrageously common in modern programming. We have explored this idea with reading from standard input, files, and other data streams. There are many other ways to connect to the outside world, one of which is via an API, or Application Programming Interface. In this context, an API refers to the functions that a server provides to allow external programs to connect and retrieve data. Understanding how an API connection works is a valuable skill to have, and we will show some examples of how to 1. connect to an API, 2. parse the data, and 3. use the data in a meaningful way.

Example 9.24. Consider a program that needs to retrieve the current weather conditions for a given latitude and longitude. Let's write a program that connects to the "OpenMeteo" Weather API, sends a request for weather data, and then parses the response.

First, we need to understand how to connect to an server-based API in Java. In general, when querying a server, we are sending what's called a GET request over HTTP (Hypertext Transfer Protocol). A GET request comprises an address, as well as parameters to tell the API what data the request asks for. In our case, we want to send a GET request to the OpenMeteo API, and we want to ask for the current weather conditions at a given latitude and longitude. The address for the OpenMeteo API is `https://api.open-meteo.com/v1/forecast`. Because we want to retrieve the weather for a given latitude and longitude case, we need to add these parameters `latitude` and `longitude` onto the end of the address. Parameters to a GET request are separated by ampersands, and preceded by a question mark. For instance, the full address to fetch the weather for Bloomington, Indiana is `https://api.open-meteo.com/v1/forecast?latitude=39.1653&longitude=86.5264&hourly=temperature_2m&timezone=EST&temperature_unit=fahrenheit`. Pasting this into a browser, we see that the browser makes the request and returns a response in the form of a JSON (JavaScript Object Notation) object, which is a common format for data transfer. To make adding additional parameters down the road easier, we will store them as strings that map to strings in a map, then concatenate them into the resulting URL string.

Now, we want to see how to do this in Java. We can create a `URL` instance with our address, and open a connection to the server by casting the `URL` to a `HttpURLConnection`. We also must tell the connection what type of request we are making, which in this case is a GET request. After establishing the connection, we must check the response code to ensure that the request was successful. Success is indicated by a response code of 200. Should the connection return a response code other than 200, we can throw some form of exception. On the other hand, if the connection succeeded, we read the response back from the server via some input stream reader. The simplest way is to use a `BufferedReader` to read the response line-by-line.

Listing 9.86

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.lang.StringBuilder;
import java.net.HttpURLConnection;
import java.net.URL;

class ApiConnection {

  public static void main(String[] args) {
    try {
      // Bind the parameters to their values.
      String webUrl = "https://api.open-meteo.com/v1/forecast?";
      Map<String, String> parameters = new HashMap<>();
      parameters.put("latitude", "39.1653");
      parameters.put("longitude", "-86.5264");

      // Create a URL with the link, then concatenate each parameter.
      StringBuilder baseUrl = new StringBuilder(webUrl);
      for (String s : parameters.keySet()) {
        baseUrl.append(String.format("%s&%s", s, parameters.get(s)));
      }

      // Open the connection and set the request to GET.
      URL url = new URL(baseUrl.toString());
      HttpURLConnection conn = (HttpURLConnection) url.openConnection();
      conn.setRequestMethod("GET");
      int resp = conn.getResponseCode();
      if (resp != 200) {
        throw new RuntimeException("main: response code "+resp+" was not 200.");
      } else {
        // Read the response from the connection line-by-line.
        try (BufferedReader br = new BufferedReader(
                                 new InputStreamReader(conn.getInputStream()))) {
          StringBuffer response = new StringBuffer();
          String inputLine = null;
          while ((inputLine = input.readLine()) != null) {
```

```java
            response.append(inputLine);
          }
        }
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

Again, the response that we receive from the API is in the JSON format, but at the moment, all we have is a large string, returned from the API, that takes the form of a JSON object. We therefore need to parse this string into a JSON object that we can then use in our program to retrieve fields and values. The question is, how do we do that? We need to take advantage of a library that can parse JSON data.[1] There are dozens to choose from, but thankfully, JSON is a simple format to understand, meaning many of the APIs are largely the same, so we shall use *JSON-java* by "stleary". We wrap our data, which we retrieved from the API, inside the constructor of a `JSONObject` instance. Doing so converts our raw string data into an object that we can traverse and access keys inside.

JSON stores its data in terms of keys and recursive values. By "recursive values," we mean that the values may, themselves, be keys to other objects. Therefore it is sensible to conclude that JSON is somewhat akin to a "multi-level" map.

Looking at the raw JSON is a good idea, since it helps us to distinguish between the important keys and values returned from the API. Passing only the latitude and longitude as HTTP parameters is not sufficient; we must also tell the API what data we want associated with that particular location. According to the open-meteo API, to view the current temperature, we must append the `current` key with the `temperature_2m` value into the request parameters. Rerunning this request now shows the data split into two distinct JSON Objects: `current_units` and `current`. The former, as its name might suggest, acts as a one-to-one mapping to the keys in `current`, where each entry associates the data with a specific unit. By default, for instance, the temperature is reported in degrees Celsius, which the API informs us. We also see the interval at which the API polls its temperature sensors/collectors: every 900 seconds, or every fifteen minutes. Now, how do we access this data in our program? We, of course, have a `JSONObject` instance, namely e, that encapsulates the raw JSON data, and we need to retrieve the temperature. From looking at the response, as we said, the temperature is a key inside the `"current"` object, which resides within e. We access this object by calling `.getJSONObject` on the JSONObject, which, itself, is a JSONObject that we can manipulate. Finally, we need to retrieve the value associated with the `"temperature"` key, that of which we know to be a `double`, i.e., a numeric temperature.

Listing 9.87

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
import org.json.JSONObject;
```

[1] We could, and is indeed a great exercise in working with real-world data and parsing techniques, write our own JSON parser.

```java
class ApiConnection {

  public static void main(String[] args) {
    try {
      // Bind the parameters to their values.
      String webUrl = "https://api.open-meteo.com/v1/forecast?";
      Map<String, String> parameters = new HashMap<>();
      parameters.put("latitude", "39.1653");
      parameters.put("longitude", "-86.5264");
      parameters.put("current", "temperature_2m");

      // Create a URL with the link, then concatenate each parameter.
      StringBuilder baseUrl = new StringBuilder(webUrl);
      for (String s : parameters.keySet()) {
        baseUrl.append(String.format("%s&%s", s, parameters.get(s)));
      }

      // Open the connection and set the request to GET.
      URL url = new URL(baseUrl.toString());
      HttpURLConnection conn = (HttpURLConnection) url.openConnection();
      conn.setRequestMethod("GET");
      int resp = conn.getResponseCode();
      if (resp != 200) {
        throw new RuntimeException("main: response code "+resp+" was not 200.");
      } else {
        // Read the response from the connection line-by-line.
        try (BufferedReader br = new BufferedReader(
                                  new InputStreamReader(conn.getInputStream()))) {
          StringBuffer response = new StringBuffer();
          String inputLine = null;
          while ((inputLine = input.readLine()) != null) {
            response.append(inputLine);
          }

          // Now that we have the data, we can parse it.
          JsonElement jsonElement = JsonParser.parseString(resp.toString());
          JsonObject jsonObject = jsonElement.getAsJsonObject();
          JsonArray times = jsonObject.get("hourly")
                                      .getAsJsonObject()
                                      .get("time")
                                      .getAsJsonArray();
          JsonArray temps = jsonObject.get("hourly")
                                      .getAsJsonObject()
                                      .get("temperature_2m")
                                      .getAsJsonArray();
        }
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

And that (which, admittedly, is a lot) is all there is to reading data from an API. This project introduces a lot of new concepts, but it opens up a whole new world of possibilities for programs; we can now connect to any (public) API and retrieve data and make decisions based on that data.

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 .88

```
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 Àssertions:" 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 .89

```
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 .90

```
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 4 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 4: 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