Joshua Crotts

Learning Java

A Test-Driven Approach

May 14, 2024

Springer Nature

To my Viola.

Preface

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

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

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

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

The ordering of topics presented in a Java course is hotly-debated and has been ever since its creation and use in higher education. Such questions include the location of object-oriented programming principles: do we start off with objects or hold off until later? Depending on the style of a text, either option can work. Even though we, personally, are more of a fan of the "early objects" approach, and is how we learned Java many moons ago, we choose to place objects later in the curriculum. We do this to place greater emphasis on testing, method design, recursion, and data structures through the Collections 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. At the high-school level, this book can be used as a substitute for the College Board's former AP Computer Science AB course. Certainly, this text may be well-suited for an AP Computer Science A course at the high-school level, but its material goes well beyond the scope of the AP exam and curriculum. Instructors are free to omit certain topics that may have been covered in a prerequisite (traditionally-styled) Java course. In those circumstances, it may be beneficial to dive further into the chapters on algorithm analysis and modern Java pragmatics. For students without a Java background (or instructors of said students), which we assume, we take the time to quickly yet effectively build confidence in Java's quirky syntax. Additionally, we understand that our approach to teaching loops (through recursion and a translation pipeline) may appear odd to some long-time programmers. So, an instructor may reorder these sections however they choose, but we strongly recommend retaining our chosen ordering for pedagogical purposes, particularly for those readers that are not taking a college class using this text.

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

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

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

Have a blast!
Joshua Crotts

Acknowledgements

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

Contents

Preface vii

1 Object-Oriented Programming 1
- 1.1 Classes 1
- 1.2 Object Mutation and Aliasing 30
- 1.3 Interfaces 51
- 1.4 Inheritance 70
- 1.5 Abstract Classes 89
- 1.6 Exercises 106

2 Exceptions and I/O 133
- 2.1 Exceptions 133
  - 2.1.1 Unchecked Exceptions 133
  - 2.1.2 Checked Exceptions 137
  - 2.1.3 User-Defined Exceptions 137
- 2.2 File I/O 138
  - 2.2.1 Primitive I/O Classes 138
- 2.3 Modern I/O Classes & Methods 149
- 2.4 Exercises 156

References 167

Index 168

1 Object-Oriented Programming

Abstract This chapter is divided into two halves. In the first half, we begin to introduce the basics of object-oriented programming. We start off simple with small classes that contain only immutable fields. The complexity gradually increases as we mix in mutability and aliasing. This chapter also describes the implementation of several data structures that mimic those from the Collections API, including `ArrayList`, `LinkedList`, `HashSet`, and maps.
In the second half, we expand on the details of object-oriented programming by describing interfaces and inheritance. Interfaces ascribe behaviors and characteristics to classes, whereas inheritance denotes "IS-A" relationships across classes. We conclude the chapter by demoing a practical application of object-oriented programming and class design by writing a small programming language.

1.1 Classes

From the first page, we have made prolific use of classes, but in this chapter, we 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. Classes encapsulate data and method definitions for later use.

As we stated, we have repeatedly used classes *and* objects, e.g., strings, arrays, `Scanner`, `Random`, as well as 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, in general, describe a noun. All Java files describe a class and must be named accordingly. We previously omitted the details of class creation and merely used them as a means to design methods.

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 design: `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 sections, we dive more into inheritance and hierarchies.

Example 1.1. Let's design the `Point` class, which stores two `int` values representing two Cartesian coordinates 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 declare a `Point` object `p`, then `p` has two instance variables, `x` and `y`, which are the x and y coordinates of `p`. If we declare another `Point` object `p2`, then `p2` has 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` are accessible only to those methods within the class definition. For the time being, instance variables are immutable. 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. Moreover, because we will not use `protected`, the use of `public` will be infrequent and only when necessary.[2]

```
class Point {

  private final int X;
  private final int Y;
}
```

We now want a way to create an instance of a `Point`. Instances of objects are made using the `new` keyword, followed by the class constructor. *Constructors* are special methods that instantiate an instance of a class. Our `Point` class constructor will receive parameterss, which we will use to initialize the relevant x and y instance variables. So, let's design the constructor for our `Point` class. Constructors, in general, should be `non-private`, as we need to call them from outside the class definition. On a case-by-case basis, this changes accordingly, as some classes are local to another class definition, and are thereby private.[3]

Constructors are also special in that they do not have an explicit return type, but they are non-void in that they return an object of the class type.

All classes in Java that can be instantiated have a *default constructor* only when no constructor is specified by the class implementer.[4] The default constructor of a class receives no arguments and serves only to be able to create an instance of the class.

[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.

[2] Some methods, as we will soon see and have seen previously, are required to be `public`. For example, the `main` method must be marked as `public`.

[3] There are also design patterns, as exemplified in Chapter **??**, that rely on a class constructor being private to prohibit unnecessary object instantiation.

[4] Later in this chapter, we will see that some classes and types cannot be instantiated.

```
class Point {

  private final int X;
  private final int Y;

  Point(int x, int y) {
    this.X = x;
    this.Y = y;
  }
}
```

Remember that the purpose of the `Point` constructor is to initialize the class instance variables. So, unless we want to use distinct identifiers for referencing the parameters and instance variables, we need to use the `this` keyword.[5] The `this` keyword refers to the current object, and aids in distinguishing between instance variables and parameters. The "current object" references the object that a method is called on.

Inside the constructor, we assign the value of the parameter `x` to the instance variable `x`. Should we 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 non-private and have strictly one purpose: to return the respective instance variable value.

```
class Point {

  private final int X;
  private final int Y;

  Point(int x, int y) {
    this.X = x;
    this.Y = y;
  }

  int getX() { return this.X; }

  int getY() { return this.Y; }
}
```

The principle of hiding the implementation details of a class and its properties is called *encapsulation*. Encapsulation is a fundamental idea 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.[6]

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/initializes a `Point` object `p`.

```
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

[5] Some software engineers and projects use identifier prefixes to refer to instance variables.

[6] By "dangerous," we mean to suggest that it is prone to logic errors.

called *object instantiation*. We use the `new` keyword and pass the desired x and y integer coordinates.

```
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 methods do nothing more than retrieve instance variable values, but it is a good habit for beginning object-oriented programmers.

```
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 method with the signature `public String toString()` (note that this is one instance where `public` cannot be avoided.) 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.

```
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.

```
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 design other methods associated with a `Point` object. For example, we might want to calculate the distance between two points. We can design a 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 receive an implicit parameter, which is the object on which the method is called, which is accessible through the `this` pointer. We say the second is *explicit* because we explicitly pass the object as a parameter. So, in the following example, `p1` is the implicit parameter, and `p2` is the explicit parameter.

```
final Point P1 = new Point(3, 4);
final Point P2 = new Point(6, 42);
double dist = p1.distance(p2);
```

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 receive parameters. Methods are functions that are associated with a class. In Java, we use the term *method* to refer to all functions, procedures, and subroutines, 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 design `distance`, which receives a `Point` as a parameter and returns the Euclidean distance from `this` point 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.

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

class PointTester {

  private 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, P1.distance(P3), DELTA),
      () -> assertEquals(5, P2.distance(P3), DELTA));
  }
}
```

```
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.
   */
  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 holistically. Static methods are useful as utility methods that are not associated with a particular instance of the class. All methods that we designed up until this chapter were static methods, which were not associated with the class in which they resided, because the classes we designed were used only to hold the methods themselves.

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 1.2. Let's overload the `distance` method by designing 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`. Namely, we pass it the origin point, i.e., `new Point(0, 0)`, and everything works wonderfully. Because this version of `distance` merely refers to the existing definition, which we have thoroughly tested, we will omit a separate set of tests.

```
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.
   */
   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 context, because a point is defined by its two coordinate members. In subsequent sections, however, we will overload other class constructors to demonstrate its utility/practicality.

Example 1.3. Let's design the static `avgDist` method, which receives a `List<Point>` and computes the average distance away each `Point` is from the origin. We already have a method

to compute the distance of a point to the origin, so let's take advantage of the stream API to map `distance` to every point, then find the average of those resulting double values. The `avgDist` method returns an `OptionalDouble` in the event that the provided list is empty, which serves as a wrapper around the `Optional<Double>` type.

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

import java.util.List;
import java.util.OptionalDouble;

class PointTester {

  @Test
  void testAvgDist() {
    List<Point> lop1 = List.of();
    List<Point> lop2 = List.of(new Point(4, 0), new Point(0, 4),
                               new Point(-4, 0), new Point(0, -4),
                               new Point(2, 2), new Point(-2, 2),
                               new Point(-2, -2), new Point(2, -2));
    assertAll(
      () -> assertEquals(OptionalDouble.empty(), Point.avgDist(lop1)),
      () -> assertEquals(3.414, Point.avgDist(lop2).get(), 0.01));
  }
}
```

```
import java.util.List;
import java.util.OptionalDouble;

class Point {
  // ... previous code not shown.

  /**
   * Computes the average distance to the origin of a list of points.
   * @param lop - list of points.
   * @return empty Optional if the list is empty,
   * or OptionalDouble otherwise.
   */
  static OptionalDouble avgDist(List<Point> lop) {
    return lop.stream()
              .map(p -> p.distance())
              .mapToDouble(d -> d)
              .average();
  }
}
```

Example 1.4. Let's design the static `random` method, which returns a `Point` object with random x and y coordinates. 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 a 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 whether the point is within a certain radius of the origin.

```
import static Assertions.assertAll;
import static Assertions.assertTrue;

class PointTester {

  @Test
  void testPointRandom() {
    assertAll(
      () -> assertTrue(Point.random(10).distance() <= 10);
      () -> assertTrue(Point.random(1).distance() <= 1);
      () -> assertTrue(Point.random(5).distance() <= 5);
      () -> assertTrue(Point.random(5000000).distance() <= 5000000));
  }
}
```

```
import java.util.Random;

class Point {

  /**
   * Generates a random point with a maximum radius.
   * @param maxRadius - the maximum radius.
   * @return a random point.
   */
  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* is a variable that is associated with the class and not a specific instance thereof. Static variables are shared among all instances of a class.

Example 1.5. Suppose that we want to count how many instances of `Point` have been instantiated in a running program. Since counting instances 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 we increment inside the constructor. To remain consistent with our recurring theme of encapsulation, `count` will be declared as `private`, and we will design a static method `getCount` to retrieve the number of instances, which is invoked on the class.[7] 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 updated inside the constructor.

[7] It is also possible to invoke a static method on an instance of the class, but it is considered bad practice and unnecessary.

```
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());
  }
}
```

```
class Point {

  private static int count = 0;

  private final int X;
  private final int Y;

  Point(int x, int y) {
    this.X = x;
    this.Y = y;
    count++;
  }

  static int getCount() {
    return count;
  }
}
```

Notice that, inside the `getCount` method, we do not refer to `count` with `this`, because `count` is a static variable and not an instance variable. Prefixing the `count` variable with `this` results in a compiler error. Further note that the variable is not marked as `final`; it is not immutable and changes with every newly-instantiated `Point` object.

Example 1.6. 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 determine if a `Point` is already inside the set?

We need to override two important methods from the `Object` class: `public boolean equals` and `public int hashCode`. The `equals` method of an object determines whether two instances of the class are "equal." In the circumstance of points, let's say that two points are equal according to `equals` only if they have 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 need to check for type equality. In other words, we first need to verify that the passed object to the `equals` method is, in fact, a `Point`, otherwise they cannot possibly be equal. To "type check" a parameter, we use the `instanceof` key-

word. If the input parameter is, indeed, a `Point`, we cast it to a `Point` instance, then check whether the coordinates match. Moreover, like `toString`, the `equals` and `hashCode` methods are definitionally `public`, so do not omit the access modifier.

The `assertEquals` and `assertNotEquals` methods invoke an object's `.equals` method when determining equality, which by default compares object references. Because we are finally overriding its implementation in `Point`, we can test two arbitrary `Point` instances for definitional equality.

```
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)));
  }
}
```

```
class Point {
  // ... previous code 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 create a `HashSet<Point>`, then iterate over the elements thereof 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 having identical 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 hash codes must also be equal. The *hash code* of an object is 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 in the `Point` class. 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 (compute the hash code of) 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 arguments. When overriding `hashCode`, we should include all instance variables of the object to designate that all of the properties affect the object's hash code. After fixing the issue, we see that our `HashSet` now correctly contains only one of the `Point` objects that we add.

```
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);
  }
}
```

```
import java.util.Objects;

class Point {
  // ... previous code not shown.

  @Override
  public int hashCode() {
    return Objects.hash(this.x, this.y);
  }
}
```

Example 1.7. Let's design the static `removeLinearPoints` method that, when given a `List<Point>`, filters out all points that are "linear," meaning that their x and y coordinates are the same. As with the previous example, streams make this exercise a walk in the park. When comparing lists, the `equals` method invokes `equals` on every element of the list, meaning that we call `Point`'s implementation of `equals`.

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

import java.util.List;

class PointTester {
  // ... previous code not shown.

  @Test
  void testRemoveLinearPoints() {
    List<Point> lop = List.of(new Point(5, 10), new Point(7, 7),
                              new Point(2, 3), new Point(4, 3),
                              new Point(1, 1), new Point(-6, -10),
                              new Point(-23, -23), new Point(1, 0));
    List<Point> lopRes = List.of(new Point(5, 10), new Point(2, 3), new Point(4, 3),
                                 new Point(-6, -10), `new Point(1, 0));
    assertAll(
      () -> assertEquals(List.of(), Point.removeLinearPoints(List.of())),
      () -> assertEquals(lopRes, Point.removeLinearPoints(lop)));
  }
}
```

```
import java.util.List;

class Point {
  // ... previous code not shown.

  /**
   * Returns a list of all points that are not "linear."
   * @param lop - list of Point objects.
   * @return list where linear points are removed.
   */
  static List<Point> removeLinearPoints(List<Point> lop) {
    return lop.stream()
              .filter(p -> p.getX() != p.getY())
              .toList();
  }
}
```

Example 1.8. Suppose that we're writing a program that keeps track of orders for a local pizzeria. Let's design the `PizzaOrder` class, which stores a `Map<String, Integer>` as an instance variable. The map associates toppings (as strings) to their respective quantities on the pizza order. Toppings can be `"Pepperoni"`, `"Onion"`, `"Pineapple"`, or `"Anchovie"`. The `PizzaOrder` constructor instantiates the map to be a `LinkedHashMap` to ensure that the order of the toppings is respected. As parameters to the constructor, it receives two arrays: `String[] toppings` and `int[] toppingCount`, where each entry is added to the map. To make the constructor simpler, we will assume that `toppings` and `toppingCount` share the same length. We'll also design the `Set<String> getToppings()` method to return the toppings in the order that they were added. Finally, we'll design a method, `Optional<Integer> getToppingCount(String s)` to retrieve the number/quantity of a given topping.[8]

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

import java.util.Optional;

class PizzaOrderTester {

  @Test
  void testPizzaOrder() {
    PizzaOrder p1 = new PizzaOrder();
    PizzaOrder p2 = new PizzaOrder(new String[]{"Pepperoni", "Anchovie"},
                                   new int[]{3, 2});
    assertAll(
      () -> assertEquals(Set.of(), p1.getToppings()),
      () -> assertEquals(Set.of("Pepperoni", "Anchovie"), p2.getToppings())
      () -> assertEquals(Optional.of(3), p2.getToppingCount("Pepperoni")),
      () -> assertEquals(Optional.empty(), p2.getToppingCount("Pineapple")));
  }
}
```

[8] The `getToppingCount` method uses `Optional.ofNullable` that, when given a `null` argument, returns the empty `Optional` and otherwise wraps the value in a non-empty `Optional`.

```
import java.util.LinkedHashMap;
import java.util.LinkedHashSet;
import java.util.Map;
import java.util.Optional;
import java.util.Set;

class PizzaOrder {

  private final Map<String, Integer> TOPPINGS;

  PizzaOrder(String[] toppings, int[] count) {
    this.TOPPINGS = new LinkedHashMap<>();
    for (int i = 0; i < toppings.length; i++) {
      TOPPINGS.put(toppings[i], count[i]);
    }
  }

  /**
   * Returns the toppings as a set.
   * @return new LinkedHashSet containing toppings.
   */
  Set<String> getToppings() {
    Set<String> set = new LinkedHashSet<>();
    this.TOPPINGS.keySet().forEach(k -> set.add(k));
    return set;
  }

  /**
   * Returns the number of a certain topping there are in this pizza order.
   * @param topping - one of the four toppings.
   * @return Optional<Integer> containing value, or empty optional.
   */
  Optional<Integer> getToppingCount(String topping) {
    return Optional.ofNullable(this.TOPPINGS.get(topping));
  }
}
```

Example 1.9. 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, i.e., what classes we need to design. It makes sense to start with a `Card` class, which stores its suit and its numeric value. A suit is one of four possibilities, each of which use a different symbol, meaning that we should 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.[9] Consequently, 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 existing as a means to support the `Card` class 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 and not an instance.

[9] Instantiating objects in this manner bears resemblance to a *design pattern* called *singleton*.

```
class Suit {

  static final Suit CLUBS = new Suit("♣");
  static final Suit DIAMONDS = new Suit("♢");
  static final Suit HEARTS = new Suit("♡");
  static final Suit SPADES = new Suit("♠");
  static final int NUM_SUITS = 4;

  private final String S_VAL;

  private Suit(String s) { this.S_VAL = s; }

  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 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 accessors and the iterator retriever.

```
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()));
  }
}
```

```
class Card {

  private final Suit SUIT;
  private final int VAL;

  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);
  }

  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, otherwise called the "face" cards. To simplify the design of our game, the face cards will be treated the same as a "ten" card, showing neither a 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, the maximum is fifty-two. In the `Deck` constructor, we call the `populateDeck` method, which adds four cards of the same value, but of each suit. So, to exemplify, 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 access.

To test a `Deck`, we can design the `drawCard` method, which retrieves the "top-most" card on the deck. 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 have the values nine, eight, and seven, all 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 `isEmpty` 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.

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

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());
  }
}
```

```
import java.util.ArrayList;
import java.util.Iterator;

class Deck {

  private static final int MAX_NUM_CARDS = 52;
  private final ArrayList<Card> CARDS;

  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.
  */
  Card drawCard() {
    if (this.CARDS.isEmpty()) {
      return null;
    } else {
      return this.CARDS.remove(this.CARDS.size() - 1);
    }
  }

  /**
   * Determines if the deck is empty.
   * @return true if the deck contains no cards, and false otherwise.
   */
  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 class, containing 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 in the previous chapter, we will once again use them to our advantage.

```
import java.util.ArrayList;

class Player {

  private final ArrayList<Card> HAND;

  Player() { this.HAND = new ArrayList<Card>(); }

  /**
   * Adds a card to the player's hand.
   * @param c - card to add to the player's hand.
   */
  void addCard(Card c) { this.HAND.add(c); }

  /**
   * Removes all cards from the player's hand.
   */
  void clearHand() { this.HAND.clear(); }

  /**
   * Determines the player's score.
   * @return the player's score.
   */
  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 design `TwentyOne`: the class that runs a game of "twenty-one." The game logic is as follows: 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 (from the deck) 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.

```
import java.util.Scanner;

class TwentyOne {

  private static final int MAX_SCORE = 21;
  private final Player PLAYER;

  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.
   */
  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() <= 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() > MAX_SCORE) {
        System.out.println("You lose!");
      } else {
        System.out.printf("You did not go over %d!", MAX_SCORE);
      }

      System.out.println("Do you want to continue playing?");
      String resp = in.nextLine();
      continuePlaying = resp.equals("Y");
    }
  }
}
```

Designing interactive games is a great exercise in object-oriented programming, as well as the culmination of other discussed topics.

Example 1.10. Suppose that we are asked to design a simple library management system for a local library. That is, the system wants to be able to check out books, determine if they are in the system, and how much stock remains. Let's think about the components of such a system. At a minimum, we need an `Author` and a `Book` class. Authors contain only a name out of conciseness. Books contain a title, author, release date, edition number, and page count. Because books contain authors, we again use object composition. Both classes, namely `Author` and `Book` are straightforward to design. Both classes also override the `equals`, `hashCode`, and `toString` methods for use in a hashable data structure and equality comparison.

```
class LibraryTester {

  private static final Author A1 = new Author("Michael Spivak");
  private static final Author A2 = new Author("Joshua Crotts");
  private static final Author A3 = new Author("Douglas Hofstader");
  private static final Author A4 = new Author("William Van Orman Quine");

  private static final Book B1
    = new Book("Calculus", A1, 4, 680);
  private static final Book B2
    = new Book("Principles of Computer Science", A2, 1, 754);
  private static final Book B3
    = new Book("Godel, Escher, Bach", A3, 2, 824);
  private static final Book B4
    = new Book("Methods of Logic", A4, 4, 344);

  @Test
  void testBook() {
    assertAll(
      () -> assertEquals("Calculus", b1.getTitle()),
      () -> assertEquals(new Author("Joshua Crotts"), b2.getAuthor()),
      () -> assertEquals(2, b3.getEditionNumber()),
      () -> assertEquals(344, b4.getPageCount()));
  }
}
```

```
class Author {

  private final String NAME;

  Author(String name) { this.NAME = name; }

  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Author)) {
      return false;
    } else {
      Author othAuthor = (Author) o;
      return this.NAME.equals(othAuthor.getName());
    }
  }

  @Override
  public int hashCode() {
    return this.NAME.hashCode();
  }
```

```
  @Override
  public String toString() {
    return this.NAME;
  }

  String getName() {
    return this.NAME;
  }
}
```

```
import java.util.Objects;

class Book {

  private final String TITLE;
  private final Author AUTHOR;
  private final int EDITION;
  private final int NUM_PAGES;

  Book(String title, Author author, int edition, int numPages) {
    this.TITLE = title;
    this.AUTHOR = author;
    this.EDITION = edition;
    this.NUM_PAGES = numPages;
  }

  @Override
  public boolean equals(Object o) {
    if (!(o instanceof Book)) {
      return false;
    } else {
      Book othBook = (Book) o;
      return this.NAME.equals(othBook.NAME)
          && this.AUTHOR.equals(othBook.AUTHOR)
          && this.EDITION == (othBook.EDITION)
          && this.NUM_PAGES == (othBook.NUM_PAGES);
    }
  }

  @Override
  public int hashCode() {
    return Objects.hashCode(this.NAME, this.AUTHOR,
                            this.EDITION, this.NUM_PAGES);
  }

  @Override
  public String toString() {
    return String.format("%s [%s]. Edition: %d. Page Count: %d",
                         this.NAME, this.AUTHOR,
                         this.EDITION, this.NUM_PAGES);
  }

  // Getters and setters omitted.
}
```

With these two classes complete, let's begin to think about the `Library` class. Let's say that it stores an alphabetized `Map<Book, Integer>` of book instances to the number of copies that are not checked out. We want to be able to add books, check books out, and determine if the book is in the library. So, let's design the `void addBook(Book b, int qty)`, `Book checkout(String title)`, `int getQuantity(String title)`, and `boolean containsBook(String title)` methods. The tests will reuse the `Author` and `Book` declarations from the previous test suite.

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

class LibraryTester {
  // ... previous test and declarations omitted.

  @Test
  void testLibrary() {
    Library l1 = new Library();
    l1.addBook(B1, 10);
    l1.addBook(B2, 1);
    l1.addBook(B3, 3);
    l1.addBook(B4, 5);
    assertAll (
      () -> assertEquals(1, l1.getQuantity("Principles of Computer Science")),
      () -> assertTrue(l1.containsBook("Methods of Logic")),
      () -> assertFalse(l1.containsBook("Frankenstein")),
      () -> assertEquals(B2, l1.checkout("Principles of Computer Science")),
      () -> assertNull(l1.checkout("Principles of Computer Science")),
      () -> assertEquals(0, l1.getQuantity("Principles of Computer Science")));
  }
}
```

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

import java.util.Map;
import java.util.TreeMap;

class Library {

  private final Map<Book, Integer> BOOKS;

  Library() {
    this.BOOKS = new TreeMap<>();
  }

  /**
   * Inserts a Book/quantity association to the map. If the book already
   * exists, we add qty to its frequency.
   * @param b - Book to add to map.
   * @param qty - quantity of provided book.
   */
  void addBook(Book b, int qty) {
    this.BOOKS.put(b, this.BOOKS.getOrDefault(b, 0) + qty);
  }
```

```
  /**
   * "Checks out" a book to someone. By checking out, we mean
   * that it searches for the book with the given title, and
   * returns its instance while decrementing its book counter.
   * If the associated counter is zero, we return null.
   * @param t - title to search.
   * @return Book instance if it exists, or null otherwise.
   */
  Book checkout(String t) {
    for (Book b : this.BOOKS.keySet()) {
      if (b.getTitle().equals(t) && this.BOOKS.get(b) > 0) {
        this.BOOKS.put(b, this.BOOKS.get(b) - 1);
        return b;
      }
    }
    return null;
  }

  /**
   * Searches through the books to determine if a book
   * with the given title exists.
   * @param t - title of book to search for.
   * @return true if a book with title t exists, false otherwise.
   */
  boolean containsBook(String title) {
    for (Book b : this.BOOKS.keySet()) {
      if (b.getTitle().equals(t)) {
        return true;
      }
    }
    return false;
  }
}
```

Example 1.11. Suppose we have the following class that represents a position on a two-dimensional board of characters:

```
class BoardPosition {

  private final char CH;

  BoardPosition(char ch;) {
    this.CH = ch;
  }

  char getChar() {
    return this.CH;
  }
}
```

Let's now design the `Board` class to represent a two-dimensional board of `BoardPosition` instances. Each instance thereof contains a `BoardPosition` that stores a single arbitrary character literal. The `Board` constructor receives integers representing the number of rows and columns of the board. It also receives a one-dimensional array of characters, which it uses to populate the board. This example helps to demonstrate how to convert a one-dimensional array into a two-dimensional array. To convert between the two, we use the formula `rows * cols + cols`.

Let's also design the `Map<Integer, List<Character>> getPredecessors()` method, which returns a map of the logical indices that precede each character in the board. That is, each key in the map is an index/position into the board, and its value is the list of characters that precede the character at that index on the board. For example, if the board contains the characters `'a', 'e', 'f', 'b', 'd', 'g', 'c'`, then the map should contain the following key-value pairs:

- `0` → `[]`
- `1` → `['a']`
- `2` → `['a']`
- `3` → `['a', 'e']`
- `4` → `['a', 'e']`
- `5` → `['a', 'e', 'f']`
- `6` → `['a', 'e', 'f']`

We note that, while this example is not particularly useful in practice, it is a good exercise in traversing a two-dimensional array and converting between one-dimensional and two-dimensional arrays.

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

import java.util.List;
import java.util.Map;

class BoardTester {

  private static final char[] BOARD = {'a', 'b', 'c', 'd', 'e', 'f'};

  @Test
  void testBoard() {
    assertAll(
      () -> assertEquals(2, new Board(2, 3, BOARD).getRows()),
      () -> assertEquals(3, new Board(2, 3, BOARD).getCols())
      () -> assertEquals('a', new Board(2, 3, BOARD).getBoard()[0][0].getChar()),
      () -> assertEquals('f', new Board(2, 3, BOARD).getBoard()[1][2].getChar()));
  }

  @Test
  void testBoardPredecessors() {
    Map<Integer, List<Character>> map
      = new Board(2, 3, BOARD).getPredecessors();
    assertAll(
      () -> assertEquals(List.of(), map.get(0)),
      () -> assertEquals(List.of('a'), map.get(1)),
      () -> assertEquals(List.of('a'), map.get(2)),
      () -> assertEquals(List.of('a', 'd'), map.get(3)),
      () -> assertEquals(List.of('a', 'd'), map.get(4)),
      () -> assertEquals(List.of('a', 'd', 'e'), map.get(5)));
  }
}
```

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

class Board {

  private final BoardPosition[][] BOARD;

  Board(int rows, int cols, char[] chars) {
    this.BOARD = new BoardPosition[rows][cols];
    for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
        this.BOARD[i][j] = new BoardPosition(chars[i * cols + j]);
      }
    }
  }

  /**
   * Returns the predecessors of each character on the board.
   * A predecessor is a character that comes before the character.
   * @return map of predecessors.
   */
  Map<Integer, List<Character>> getPredecessors() {
    Map<Integer, List<Character>> map = new HashMap<>();
    for (int i = 0; i < this.BOARD.length; i++) {
      for (int j = 0; j < this.BOARD[0].length; j++) {
        int index = i * this.BOARD[0].length + j;
        List<Character> list = new ArrayList<>();
        for (int k = 0; k < index; k++) {
          int row = k / this.BOARD[0].length;
          int col = k % this.BOARD[0].length;
          list.add(this.BOARD[row][col].getChar());
        }
        map.put(index, list);
      }
    }
    return map;
  }
}
```

Example 1.12. Let's design the `Rational` class, which stores a rational number as a numerator and denominator. We will create methods for adding, subtracting, multiplying, and dividing rational numbers. Testing is paramount to designing a correct implementation, as is the case with all projects, but is of particular significance here. 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 gloss over the actual mathematics and prioritize the Java implementation and class design.

The `Rational` constructor receives two integers p and q, and assigns them as instance variables. The `toString` method only involves placing a slash between our numerator and denominator. Though, let's back up for a second and think about 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/reduces the fraction into its lowest terms. Fractions can be reduced 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.

```
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()));
  }
}
```

```
class Rational {

  private final long NUMERATOR;
  private final long DENOMINATOR;

  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 need to 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 multiply 2 by 9 and 7 by 3 to get 18/27 and 24/27. Adding across the numerators produces 42/27, which then 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).

```
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()));
  }
}
```

```
class Rational {
  // Other details not shown.

  /**
   * Adds two rational numbers.
   * @param r - the other rational number.
   * @return the (simplified) sum of this and r.
   */
  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 its correspondence to addition, we leave `subtraction` as an exercise to the reader. We can now implement multiplication, which is even simpler than addition; all that is needed is to multiply the numerators and denominators together. We also leave `division` as an exercise to the reader. We 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 (rather than `long` types), 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 and of itself.

```
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()));
  }
}
```

```
class Rational {
  // ... previous code not shown.

  /**
   * Multiplies two rational numbers.
   * @param r - the other rational number.
   * @return the (simplified) product of this and r.
   */
  Rational multiply(Rational r) {
    return new Rational(this.NUMERATOR * r.NUMERATOR,
                        this.DENOMINATOR * r.DENOMINATOR);
  }
}
```

Example 1.13. 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 mechanically 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, which is 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, where there is orders of magnitude more memory space. This solution is neither fast nor space-efficient, but serves to show that naturally standard recursive algorithms, e.g., factorial, can still use a standard recursion algorithm, in a sense.

To create our own stack, we first need to decide what to place onto 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, because all classes *are* a "kind of" `Object`. 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 design the `addParam` method that receives a `String` and an `Object`, enters those into the existing map, and returns the existing instance.[10] 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.

```
import java.util.HashMap;
import java.util.Map;

class StackFrame {

  private Map<String, Object> PARAMS;

  StackFrame() {
    this.PARAMS = new HashMap<>();
  }

  Object getParam(String s) {
    return this.PARAMS.get(s);
  }

  StackFrame addParam(String s, Object o) {
    this.PARAMS.put(s, o);
    return this;
  }
}
```

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 the recursive definition, we design the `factLoop` method that instantiates a `Stack<StackFrame>` to replicate the call stack. We begin the process by pushing the initial frame comprised of 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.

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

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
```

[10] This idea resembles the *builder* design pattern, which we will discuss in Chapter **??**.

```
  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"))));
  }
}
```

```
import java.util.Stack;
import java.util.BigInteger;

class StackFrameDriver {

  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)));
    }
  }

  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.

```
class StackFrameDriver {
  // ... previous code not shown.

  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;
  }
}
```

Notice two things: first, we mimic the behavior of the call stack manually. Consequently, 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. Of course, that 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` error.

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.[11]

1.2 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 discuss the implications of instance variable mutation, and how it can lead to unintended problems, but also how it can be used to our advantage.

To access a `private` instance variable, we design a non-`private` accessor method, which returns the instance variable. To *modify* a `private` instance variable, we design a non-`private` mutator method, which receives a parameter and assigns its value to the corresponding instance variable. Let's return to the `Point` class to demonstrate. Suppose that we instantiate a `Point` object p to the position $(7, 4)$, but we then wish 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 is confirmed through the accessor method.

Another way of phrasing such an approach 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 alter an instance variable, these can no longer be marked as `final`, so we remove the keyword.[12]

```
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());
  }
```

[11] On an AMD Ryzen 5 3600 with 16GB of DDR4 RAM, this test did not complete within a three hour time frame.

[12] 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.

```
  @Test
  void testSetY() {
    Point p = new Point(7, 4);
    p.setY(2);
    assertEquals(2, p.getY());
  }
}
```

```
class Point {

  private int x;
  private int y;

  Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  int getX() { return this.x; }
  int getY() { return this.y; }
  void setX(int x) { this.x = x; }
  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 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 is also affected.

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

class PointTester {

  @Test
  void testPointAliasing() {
    Point p1 = new Point(7, 4);
    Point p2 = p1;
    p1.setX(3);
    assertAll(
      () -> assertEquals(3, p1.getX()),
      () -> assertEquals(3, p2.getX()),
      () -> assertEquals(p1, p2));
  }

  @Test
  void testSetX() {
    Point p1 = new Point(11, 13);
    Point p2 = p1;
    p2.setX(100);
    assertAll(
      () -> assertEquals(100, p1.getX()),
      () -> assertEquals(100, p2.getX())
      () -> assertEquals(p1, p2));
  }
```

```
  @Test
  void testSetY() {
    Point p1 = new Point(7, 4);
    Point p2 = p1;
    p1.setY(2);
    assertAll(
      () -> assertEquals(2, p1.getY()),
      () -> assertEquals(2, p2.getY()),
      () -> assertEquals(p1, p2));
  }
}
```

Aliasing carries over to other, more complex classes as well. For example, strings, arrays, lists, and others are all objects, and therefore, are subject to object aliasing. Modifying one `ArrayList` instance will modify all other `ArrayList` instances that reference the same object. Unintentional aliasing (or its associated actions) 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. 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, in fact, contain references to the same `Point` instances.

```
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 testPointArrayListAliasing() {
    List<Point> list1 = new ArrayList<>(List.of(P1, P2, P3));
    List<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 1.14. Now that we have classes, accessibility, and mutation, we can implement generic data structures such as an `ArrayList`. In this example, we will design a class that matches the behavior of the `ArrayList` class. Let's design the `MiniArrayList` class, 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 methods observe/respect the quantify and do not need to be separately quantified. Static methods still necessitate the generic quantifier in their signatures.[13]

In addition to the class header, what else does an `ArrayList` store? Certainly, a backing array of elements and its corresponding length. The array, as we described in Chapter **??**, "dynamically resizes" as we add or insert elements. The logical size of the array, i.e., the number of presently-existing elements is its `size`, whereas the current capacity, i.e., how many elements can currently be stored without a resize, is its `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 the initial capacity. Interestingly, this shows off a great example of one constructor calling another of the same class, an idea called *constructor chaining*. "Ten," however, is a magic number: a number whose context is the only thing that determines its meaning. So, let's refactor it into a constant class variable with a relevant identifier.

```
class MiniArrayList<T> {

  private static final int DEFAULT_CAPACITY = 10;

  private T[] elements;
  private int size;
  private int capacity;

  MiniArrayList() {
    this(DEFAULT_CAPACITY);
  }

  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 we immediately encounter is that we cannot instantiate an array of a generic type, because Java arrays utilize runtime information about the element type. Generics, on the other hand, are a compile-time feature, meaning it is impossible to directly instantiate an array of a generic type. Instead, we must instantiate an array containing (elements of) type `Object`, followed by a cast to contain (elements of) type `T`.[14] This is called an *unchecked cast*, and it is a necessary evil in Java to support powerful classes that operate over generic arrays.

[13] Static methods are not tied to any instance of the class (nor its generic type), so they must be quantified separately.

[14] To be pedantic, the array is of type `Object[]`, and we cast it to type `T[]`.

```
class MiniArrayList<T> {

  private static final int DEFAULT_CAPACITY = 10;

  private T[] vals;
  private int size;
  private int capacity;

  MiniArrayList() {
    this(DEFAULT_CAPACITY);
  }

  MiniArrayList(int capacity) {
    this.size = 0;
    this.capacity = capacity;
    this.vals = (T[]) new Object[this.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, 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 common practice.[15] We make `resize` private because it is an implementation detail that the programmer who uses `MiniArrayList` should not concern themselves over.

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 consider invalid inputs, e.g., negative array indices, and all inputs to methods are assumed to be semantically correct.

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

class MiniArrayListTester {

  @Test
  void testAdd() {
    MiniArrayList<Integer> list = new MiniArrayList<>();
    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)));
  }
}
```

[15] Another logical choice is to increase the capacity linearly based on the default capacity. This choice removes the need for an extra static variable.

```
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.
   */
  void add(T element) {
    if (this.size == this.capacity) {
      this.resize();
    }
    this.vals[this.size++] = element;
  }

  /**
   * Retrieves an element at a given index. The index should
   * be in-bounds. If not, an ArrayIndexOutOfBoundsException is thrown.
   * @param index - list index between 0 <= i < L.size()
   * @return item at the given index.
   */
  T get(int index) {
    return this.vals[index];
  }

  /**
   * Returns the logical size of the list, i.e., the number of
   * actual elements in the list.
   * @return number of elements.
   */
  int size() {
    return this.size;
  }

 /**
  * 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.vals[i];
    }
    this.vals = newArray;
 }
}
```

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 new test cases. Like the `get` counterpart, neither of these new methods will perform bounds checking, so testing out-of-bounds behavior is not pertinent.

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

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)));
  }
}
```

```
class MiniArrayList<T> {
  // ... previous code not shown.

  /**
   * Inserts an element at the given index.
   * @param e - the element to insert.
   * @param idx - the index to insert at.
   */
  void insert(T e, int idx) {
    if (this.size == capacity) {
      this.resize();
    }
    this.shiftRight(idx);
    this.vals[idx] = e;
  }

  /**
   * Removes the element at the given index.
   * @param idx - the index to remove.
   * @return the element removed.
   */
  T remove(int idx) {
    T e = this.get(idx);
    this.shiftLeft(idx);
    this.size--;
    return e;
  }
}
```

```
class MiniArrayList<T> {

  /**
   * 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.vals[i] = this.vals[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.vals[i] = this.vals[i - 1];
    }
  }
}
```

Example 1.15. 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.

Let's consider 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.

```
import static Assertions.assertEquals;
import static Assertions.assertNotEquals;

class ClassTester {

  @Test
  void testOne() {
    final A a = new A("Hello!");
    B b = new B(a);
    C c = new C(a);
    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.

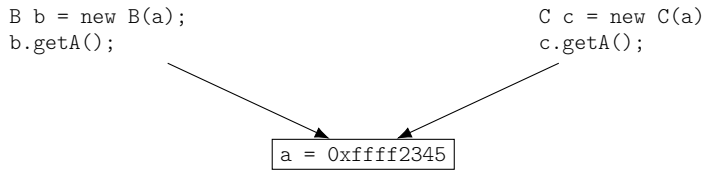

Fig. 1.1: Memory Aliasing Example

In Figure 1.1, 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.

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.

Example 1.16. Let's do another aliasing test, this time to involve arrays of objects. We will operate over a class `E` that stores a single number, similar to how the `Integer` class works.

```
class E {

  private int val;

  E(int v) { this.val = v; }

  int getNumber() { return this.val; }

  void setNumber(int v) { this.val = v; }
}
```

```
import static Assertions.assertEquals;
import static Assertions.assertNotEquals;

class ClassTester {

  @Test
  void testTwo() {
    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 object `e` is instantiated to a new instance of `E`, whose constructor receives 42 as an argument. Thereafter we instantiate `arrOfE` to be an array of ten `E` objects. The following loop then instantiates each index of the array to a new, distinct `E` object with the integer `i` as an argument to the constructors.

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. Each E instance receives a different value of i, entailing that the equality does not hold.

The second loop assigns the e object to each index of the array. We can then compare any arbitrary element against any other arbitrary element, and they will always be equal according to both == and equals, since every element is a reference to the same memory reference. Thus, changing the stored integer value at one index propagates to every other element in the array, because again, all references point to the same object. Preceding the final assertion is an assignment of a new E instance to index four that boxes the integer 2. Changing any index aside from four modifies e, but does not modify the instance of E at index four, the converse of which also holds.

Example 1.17. Let's get even more practice with aliasing over arrays. Suppose we have the following code segment:

```
class ArrayAliasingTester {

  static int baz(int[] A) {
    A[3] = 42;
    return A[3];
  }

  static int modify(int[] A) {
    A = new int[100];
  }

  @Test
  void testBaz() {
    int[] A = new int[]{0, 0, 0, 0, 0};
    A[2] = baz(A);
    assertEquals(84, A[2] + A[3]);

    int[] B = new int[]{0, 0, 0, 0, 0, 0};
    int[] C = B;
    assert______(84, C[3] + baz(B));

    int[] D = new int[]{0, 0, 0, 0, 0, 0, 0};
    int[] E = D;
    int res = baz(E);
    assert______(84, D[3] + res);
    D = new int[]{1, 2, 3, 4, 5, 6, 7};
    assert______(E, D);

    int[] F = new int[]{0, 1, 2, 3, 4};
    int h1 = F.hashCode();
    modify(F);
    assertEquals(h1, F.hashCode());
  }
}
```

Again, we encourage the readers to stop and think about what kinds of assertions to fill in the blanks. Consider what happens when arrays are passed to methods, then continue onward to check your understanding.

We start by examining the baz method, which receives an array of integers A as a parameter, overwrites the value at index 3 to be 42, then returns that element.

Inside the JUnit tester method, we first instantiate an array A to contain five zeroes. Afterwards, we assign, to index 2 of A, the value of invoking `baz(A)`, meaning `A[2]` is 42, but so is `A[3]`, because passing an array to a method passes a copy of the reference to the array. Therefore, the index of the passed array is mutated. We assert whether the sum of these two elements is 84, which is true because $42 + 42 = 84$.

Second, we instantiate an array B to contain six zeroes. This is followed by an initialization of an array C to point to B, meaning C is an alias for B. We then assert whether the sum of `C[3]` and `baz(B)` is 84. The answer relies on an understanding of evaluation order. The plus operator evaluates its arguments from left-to-right. Before invoking `baz(B)`, the value of `C[3]` is zero, so the left-hand side of the addition is zero. Immediately after, we evaluate `baz(B)`, which mutates not only index 3 of B, but also index 3 of C, because again, C aliases B. The expression is now an addition of 0 and 42, which is certainly not equal to 84.

Third, we instantiate yet another array E to contain seven zeroes. We then alias the array D to E, followed by a call to `baz(E)`, whose result is stored in the `res` variable. We know for certain that `res` contains 42, and we want to know whether `D[3]` also contains 42 to sum to the expected value of 84. Because D aliases E, it must be the case that `D[3]` is also 42, meaning we assert equals.

The fourth assertion is the result of altering D's reference. Recall that D initially references the new integer array of seven zeroes. The D array becomes an alias to E, which might suggest that changing E also changes what D points to, but this is not the case. When we instantiate E to point to the new array of the integers from 1 to 7, D remains aliased to the array of seven zeroes that E was instantiated to. Therefore, the assertion should be not equals.

Finally, we instantiate an array F to contain the integers from 0 to 4 inclusive. We compute the hash code of F and store it inside h_1. Then, we pass F to the `modify` method, which appears to modify the passed reference to point to a new array. In the previous chapter we mentioned "pass by pseudo-reference," which alluded to this problem. When we pass an object to a method, we pass a copy *of* the reference rather than the reference itself. Changing what the copy points to does not change the reference outside the context of the `modify` method. So, we should assert equals on h_1 and the hash code of F after invoking `modify`, since F's reference is never altered.

Example 1.18. Some readers may question why we emphasize mutation and aliasing. When working with the Collections API and designing data structures, proper care must be taken to avoid undesired behavior and outcomes. Consider what happens if we design a class `F` whose constructor receives a `List<Integer>`, which is assigned directly to an instance variable. Then, suppose we instantiate two distinct instances of `F`, namely f_1 and f_2, each of which receive the same (reference to a) list of numbers. If we then mutate the list somewhere inside of f_1, then the list stored as a reference inside f_2 also contains the change.

```
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));
  }
}
```

```
import java.util.List;

class F {

  private final List<Integer> LS;

  F(List<Integer> ls) { this.LS = ls; }

  List<Integer> getList() { return this.LS; }
}
```

Example 1.19. Recall the `LinkedList` class from Chapter **??**. 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: it is 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 successor *and* their predecessor.

We need a generic class that stores references to the first and last elements of the list. Let's design 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 (and should not) be exposed to the innards of the class, i.e., how the links are established/constructed/altered/removed. After all, we wish to preserve the encapsulation motif.

We run into an imminent 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 is nonsense to say that a private class definition belongs to an arbitrary instance of the class in which it resides.[16] We also override the `toString` method to output a stringified representation of underlying node data.

```
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> first;
  private Node<T> last;

  DoublyLinkedList() { this.last = this.first = null; }
}
```

[16] Such a claim would imply that every instance of the `DoublyLinkedList` class would carry the data for instantiating nodes, which is wasteful.

Notice that, in the constructor of `DoublyLinkedList`, we assign the first and last references to each other, both of which point to `null`. An empty list contains neither a first nor a 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. There is no desire to expose the implementation of the iterator to the caller; they are only concerned with iterating over the doubly-linked list.

```
import java.util.Iterator;

class DoublyLinkedList<T> {
  // ... previous code not shown.

  Iterator<T> iterator() {
    return new DoublyLinkedListIterator<>(this.first);
  }

  private static class DoublyLLIterator<T> implements Iterator<T> {

    private Node<T> current;

    private DoublyLLIterator(Node<T> first) {
      this.current = first;
    }

    @Override
    boolean hasNext() {
      return this.current != null;
    }

    @Override
    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`.

```
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()));
  }
}
```

```
class DoublyLinkedList<T> {
  // ... previous code not shown.

  /**
   * Adds a new node to the end of the list.
   * @param data - The data to be stored in the new node.
   */
  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`.[17]

[17] It is, in general, a better idea to use *exceptions* when encountering bad inputs, but we have not covered them at this point in the text.

```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)));
  }
}
```

```java
import java.util.Optional;

class DoublyLinkedList<T> {
  // ... previous code not shown.

  /**
   * 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.
   */
  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();
  }
}
```

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, and is eventually consumed/reclaimed by the garbage collector.
(d) If the element-to-remove e is not in the list, do nothing.

```
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()));
  }
}
```

```
class DoublyLinkedList<T> {
  // ... previous code not shown.

  /**
   * Removes an element from the linked list, if it exists.
   * @param data - value to be removed, compared via .equals.
   */
  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;
        } else if (curr == this.last) { // Case 2: if it's the last.
          curr.prev.next = null;
          this.last = curr.prev;
        } else { // Case 3: if it's anything else.
          curr.prev.next = curr.next;
          curr.next.prev = curr.prev;
        }
        break;
      } else { curr = curr.next; }
    }
  }
}
```

Example 1.20. Some programming languages, e.g., C, do not come standard with data structures such as a map. A substitute for the common mapping data structure is called an *association list*, originating with the Lisp programming language McCarthy (1962). 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, 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 `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. We want to preserve the idea of immutability, which is a common theme in functoinal programming. Association lists, therefore, need to have a "parent" reference to keep track of those associations in the list that we extend from.[18] We will also override the `toString` method to print the associations in a readable format.

```
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),
      () -> 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")));
  }
}
```

[18] In the next section on abstract classes and interpreters, we will revisit this idea in greater detail.

```java
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;

  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.
   */
  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.
   */
  AssociationList<K, V> extend(K key, V value) {
    return new AssociationList<>(key, value, this);
  }
```

```java
  @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 and lookup associations, at least when compared to data structures such as a `HashMap` or `TreeMap`. But, association lists are commonly used for adding (variable) bindings in a programming language, usually when writing simple interpreted languages.[19]

Example 1.21. In Chapter **??**, we began to explore and use the `Set` data structure from the Collections API. One existing implementation of sets is the `HashSet`, which stores non-duplicate elements according to their hash codes. In this example we will design a related data structure called a *hash table*.

Elements in a hash table are stored according to their hash code. The properties of a hash table (and the hashing function) guarantee very fast element lookup times, insertions, and removals. A straightforward hashing function is to modulo the hash code of the object by the capacity of the table. This way, every element is guaranteed a valid index into the table. The problem with this approach is that multiple objects may hash to the same "bucket." For example, if the hash code of some object o_1 is 100, and the capacity of the table is 40, then we would store o_1 at index $100 \bmod 40 = 2$. But, consider another object o_2 with the hash code 1000, which we store at index $1000 \bmod 40 = 2$. We cannot store multiple objects at the same index, so we need to perform *collision resolution*. A collision, as we just demonstrated, occurs when two objects hash to the same index.

As noted in our initial discussion of hashable data structures, there are several algorithms for resolving hash collisions. The simplest, albeit the least performant, is *chaining*, where each index of the hash table stores a list of elements. If two elements hash to the same index, then we walk the list at that index to query the hash table.

Let's begin by designing the constructor, fields, and method skeletons for the generic `HashTable` class. It stores an array of list instances, and the constructor instantiates this array to a capacity specified as a static class variable. We will include a second constructor to allow the user of the class to specify a capacity. Our hash table will not be resizable at runtime.

```
import java.util.LinkedList;
import java.util.List;

class HashTable<T> {

  private static final int DEFAULT_CAPACITY = 50;

  private final List<T>[] ELEMENTS;

  private int size;

  HashTable(int capacity) {
    this.ELEMENTS = new List<>[capacity];
    for (int i = 0; i < this.ELEMENTS.length; i++) {
      this.ELEMENTS[i] = new LinkedList<>();
    }
  }

  HashTable() { this(DEFAULT_CAPACITY); }
```

[19] This serves as foreshadowing for chapter 1!

```
  /**
   * Adds a value to the hash table. We compute its hash code, then
   * insert it onto the end of the bucket at that index in the table.
   * If the value is already in the hash table, we return false and do
   * not add it into the table.
   * @param v - value to add.
   * @return true if the element ∉ hash table; false otherwise.
   */
  boolean add(T v) { /* TODO. */ }

  /**
   * Determines whether an element exists in the hash table. We compute
   * the hash code of the input parameter, then traverse the bucket at
   * that index.
   * @param v - value to search for.
   * @return true if the element ∈ hash table; false otherwise.
   */
  boolean contains(T v) { /* TODO. */ }

  int size() { return this.size; }
}
```

The add method relies on contains, so we will design the latter first. We begin by computing the hash code of the given argument, then clamp that value to the bounds of the underlying array.[20] Every index in the table corresponds to a LinkedList, so we traverse the list in search of the element and return whether or not it exists.

Designing add is almost identical to contains: we compute the index-to-insert, traverse the list, and if the element is not in that list, we add it to the end and return true. Otherwise, the element is already in the table and we return false. Upon successfully adding an element to the table, we increment the size instance variable.

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

class HashTableTester {

  @Test
  void testHashTable() {
    HashTable<String> t = new HashTable<>();
    assertAll(
      () -> assertTrue(t.add("Tarski")),
      () -> assertTrue(t.add("Quine")),
      () -> assertTrue(t.add("Russell")),
      () -> assertTrue(t.add("Boole")),
      () -> assertFalse(t.add("Tarski")),
      () -> assertTrue(t.contains("Quine")),
      () -> assertFalse(t.contains("Carnap")),
      () -> assertEquals(4, t.size()));
  }
}
```

[20] Java's modulo operator may return a negative number, so we wrap the operation in a call to Math.abs.

```
class HashTable<T> {
  // ... other methods not shown.

  /**
   * Adds a value to the hash table. We compute its hash code, then
   * insert it onto the end of the bucket at that index in the table.
   * If the value is already in the hash table, we return false and
   * do not add it.
   * @param v - value to add.
   * @return true if the element ∉ hash table; false otherwise.
   */
  boolean add(T v) {
    if (this.contains(v)) { return false; }
    else {
      int idx = Math.abs(v.hashCode() % this.BUCKETS.length);
      List<T> bucket = this.BUCKETS[idx];
      bucket.add(v);
      this.size++;
      return true;
    }
  }

 /**
  * Determines whether an element exists in the hash table. We compute
  * the hash code of the input parameter, then traverse the bucket at
  * that index.
  * @param v - value to search for.
  * @return true if the element ∈ hash table; false otherwise.
  */
  boolean contains(T v) {
    int idx = Math.abs(v.hashCode() % this.BUCKETS.length);
    List<T> bucket = this.BUCKETS[idx];
    return bucket.stream()
                 .anyMatch(t -> t.equals(v));
  }
}
```

1.3 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 we want to be able to inhibit "comparable" behavior. In particular, there is a single method that must be implemented by any class that implements the `Comparable` interface: `compareTo`. The `compareTo` method receives a single parameter of the same type as the class that implements the `Comparable` interface, and returns an integer. Said integer is negative if `this` object instance is less than the passed argument, zero if `this` object instance is equal to the passed argument, and positive if `this` object instance is greater than the passed 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 "kinds of" `List` objects, and this interface describes several methods that all lists, by definition, must override.[21] 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 1.22. 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 their names with `I` to distinguish them from classes. Moreover, the names of interfaces are either adjectives or, more traditionally, verbs, since they describe behaviors or characteristics of a class.[22]

```
interface IShape {

  double area();

  double perimeter();
}
```

We cannot write any tests for the `IShape` interface directly, because it is impossible to instantiate an interface. As defined, interfaces are a way of grouping classes by behavior. It, therefore, does not make sense to be able to instantiate an interface, because that would suggest that the interface in and of itself is an object. We can, however, write two different classes that implement `IShape`, and test those classes. To demonstrate this concept, we will design 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` instead of `Pentagon` or `Octagon`. We want to be able to categorize these classes as types of `IShape` instances rather than solely instances of `Pentagon` or `Octagon` respectively. Instantiating a variable as an interface type, then instantiating it as a subtype is a form of *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.

When implementing the methods of an interface in a class, we must mark those methods as `public` because all interface methods are `public`, either explicitly or implicitly. In this context, the `area` and `perimeter` methods are overridden in the `Octagon` and `Pentagon` classes.

[21] Here we clarify that "kind of," in this context, means to implement the `List` interface.

[22] We do not add the `public` keyword to the interface definition nor any methods within because all interface methods are implicitly public.

```
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));
  }
}
```

```
class Pentagon implements IShape {

  private final double SIDE_LENGTH;

  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;
  }
}
```

```
class Octagon implements IShape {

  private final double SIDE_LENGTH;

  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 1.23. Recall from the previous chapter our "Twenty-one" card game example. In that small project, we designed the `Suit` class, which encapsulated four static instances of `Suit`, where each represented one of the four valid card suits. Even though 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, thereby requiring any implementing class to override the `stringify` method.

```
interface ISuit {

  /**
   * Returns the string representation of the suit.
   */
  String stringify();
}
```

From here, we define four separate classes, all of which implement `ISuit` and override the `stringify` method. These classes are incredibly simple, and as such, we show only the `Diamond` and `Heart` classes.

```
class Diamond implements ISuit {

  Diamond() {}

  @Override
  public String stringify() { return "♢"; }
}
```

```
class Heart implements ISuit {

  Heart() {}

  @Override
  public String stringify() { return "♡"; }
}
```

As shown, 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.

```
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 is significant, suppose we initialize an object s_1 to be of type `ISuit`, but instantiate it as type `Diamond`. Then, we initialize another object s_2 to be of 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 invoke `diamondCount` on the s_1 object, 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 1.24. 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 have the capability of "speaking" together. We can follow this by designing classes to implement the `IAnimal` interface, which provide an implementation of the `speak` method. When testing these classes, we can instantiate a collection of `IAnimal` instances, and invoke `speak` on each of them polymorphically. In doing so, we debut a refresher of the stream API.

```
interface IAnimal {

  /**
   * Returns the sound that the animal makes.
   */
  String speak();
}
```

```
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()));
  }
}
```

```
class Cat implements IAnimal {

  @Override
  public String speak() { return "Meow!"; }
}
```

```
class Dog implements IAnimal {

  @Override
  public String speak() { return "Woof!"; }
}
```

Example 1.25. Suppose we want to design an interface that "boxes" an arbitrary value. We have seen this idea through Java's autoboxing and autounboxing mechanisms of the primitive datatypes via the wrapper classes. Our interface, however, extends the concept to any type. We can define an interface that requires any class to implement 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, in Java, we pass primitives to methods by value and, therefore, any changes to the argument are not preserved outside the method body. If, however, we box the primitive, the box is passed by reference, and it is then possible to manipulate the contents of the box. We will first design the generic `IBox` interface, then we will design a class that implements the methods specified by the interface.

Interestingly, interfaces can contain static methods. Our `IBox` interface has the static `box` method, which returns a box that encapsulates the provided value. The `box` method can be used without having to instantiate a class that implements the `IBox` interface. From there, we can write the `get` and `set` methods to retrieve and change the value of the box.

```
interface 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);
}
```

```
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()));
  }
}
```

```
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;
  }
}
```

Example 1.26. 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 such as `Scanner`, which waits for the user to press "Enter" when they are finished typing input, the Swing API allows for dynamic input and whose events are processed as they occur. We call the part of the program that listens and processes events an *event listener*. A popular example in Java is the `ActionListener` interface, which is used to listen for a broad classification of events, ranging from button clicks to menu selections. When an event occurs, the `ActionListener` interface is notified and can then respond to the event however the programmer desires. 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 1.27. 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 methods: the concept in which methods and data are equivalent, wherein both can be passed to and returned from methods. In Java, we can pass methods around as arguments, mimicing first-class methods, 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`, which 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. Then, by passing the class around as an argument to other methods, we can invoke the `apply` method on the class to get the method result. An incredibly simple example is `AddOne`, which implements the `Function<Integer, Integer>` interface and adds one to its input. We mark the constructor of the implementing class as private to prevent any unnecessary instantiations; the class itself should only ever be utilized as a first-class citizen rather than an object.

```
interface Function<T, V> {

  static V apply(T t);
}
```

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

class AddOneTester {

  @Test
  void addOneTester() {
    assertAll(
      () -> assertEquals(0, apply(1)),
      () -> assertEquals(3, apply(2)),
      () -> assertEquals(30001, apply(30000)));
  }
}
```

```
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 methods in Java with our design. Suppose that l is a list of `Integer` values $v_1, v_2, ..., v_n$ and f : `Integer` $\rightarrow$ `Integer`. We want to apply f to each element of the list l and produce a new list that is the result of mapping f over l. That is, we will create a new list $l' = f(v_1), f(v_2), \ldots, f(v_n)$. By passing a class that implements a functional interface as a parameter, we can design a single method that receives a list and a function f, rather than having to redundantly design several methods to work over multiple variants of f. This operation, in the functional programming domain, is called `map`, which we saw during our discussion on streams in Chapter **??**.[23]

[23] Do not confuse this with the concept of a map/dictionary from our data structures/collections discussion.

```
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),
                         map(l, addOne)),
      () -> assertEquals(List.of(),
                         map(List.of(), addOne)));
  }
}
```

```
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.
   */
  static <T, V> List<V> map(List<T> l, Function<T, V> f) {
    return l.stream()
            .map(t -> f.apply(t))
            .collect(Collectors.toList());
  }
}
```

Example 1.28. Java 8 introduced the `Function` interface, so we do not have to design our own version. When using it, we do not need to design a separate `AddOne` class to implement the interface; we can instead opt to use method referencing via the `::` operator. Let's rewrite the `addOne` example doing so. Concurrently, we will show off the fact that Java autoboxes and unboxes primitives into wrapper class counterparts in the functional interface, meaning that our `addOne` method does not need to receive and return objects, but rather primitives, which are easier to work with. Moreover, lambda expressions are passable to methods that receive `Function` arguments, since Java automatically converts them into `Function` objects, mimicing the autoboxing treatment of primitive datatypes.[24]

[24] 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.

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

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 1.29. Now that we have interfaces, we can write a very simple expression tree interpreter! What do we mean by this? Consider the arithmatic 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 the notion of evaluation trees relate to interfaces? Suppose we create the `IExpr` interface, which encompasses the `int value` method to resolve to the value of an expression.

```
interface IExpr {

  /**
   * Returns the value of the expression.
   */
  int value();
}
```

The simplest (atomic) values in our language are numbers, or numeric literals. A `Lit` stores a single integer as an instance variable, and returns this instance variable upon a `value` invocation, which means `Lit` should implement the `IExpr` interface. Testing this class is trivial, so we will only write two tests.

```
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()));
  }
}
```

```
class Lit implements IExpr {

  private final int N;

  Lit(int n) { this.N = n; }

  @Override
  public int value() { return this.N; }
}
```

How do we add two numbers? Or, rather, how do we represent the addition of two (literal) numbers? This question comes via the answer to our other question of representing literal values. Additive expressions store two `IExpr` objects as instance variables, and (mutually) recursively calls their `value` methods, followed by a summation to those results. Note the parallelism to how we do this when manually evaluating, say, parenthesized addition expressions.

```
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)))));
  }
}
```

```
class Add implements IExpr {

  private final IExpr LHS;
  private final IExpr RHS;

  Add(IExpr lhs, IExpr rhs) {
    this.LHS = lhs;
    this.RHS = rhs;
  }

  @Override
  public int value() {
    return this.LHS.value() + this.RHS.value();
  }
}
```

Thus, we now 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 1.30. Symbolic differentiators are programs that take a mathematical expression and compute its derivative, but non-numerically. That is, symbolic differentiators examine and interpret the structure of an expression to calculate its 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 reasonably follow along and understand the high-level and pertinent object-oriented details.

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 served as 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(sub-)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 sub-components, we obtain the derivative of the larger, similar to our arithmetic expression resolver. Let's see what all we need to do.

First, let's design the `Expression` interface, which houses one method, to compute the derivative of an `Expression`: `Expression derivative(String v)`. Any class that implements `Expression` must override `derivative`. We differentiate expressions with respect to a given variable, e.g., x, so we need to pass that variable to any expression that we wish to differentiate.

Using some basic calculus derivative shortcuts/rules, we can easily think of two more types of expressions: numeric 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 v, 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 because the monomial does not depend on the variable v.

```
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")));
  }
}
```

```
import java.util.Objects;

class ConstantExpression implements Expression {

  private final int CONSTANT;

  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 {
      ConstantExpression cons = (ConstantExpression) obj;
      return cons.CONSTANT == this.CONSTANT;
    }
  }

  @Override
  public int hashCode() {
    return Objects.hash(this.CONSTANT);
  }

  @Override
  public String toString() {
    return String.format("%d", this.CONSTANT);
  }
}
```

```
import java.util.Objects;

class MonomialExpression implements Expression {

  private final int COEFFICIENT;
  private final int EXPT;
  private final String VAR;

  MonomialExpression(String v, int a, int n) {
    this.VAR = v;
    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, i.e., those 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.

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

class DerivativeTester {
  // ... previous 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")));
  }
}
```

```
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;

  AddExpression(Expression... exprs) {
    this.EXPR_LIST = Arrays.asList(exprs);
  }

  AddExpression(List<Expression> exprs) {
    this.EXPR_LIST = exprs;
  }

  @Override
  public Expression derivative(String v) {
    List<Expression> exprs = new ArrayList<>();
    this.EXPR_LIST.forEach(e -> exprs.add(e.derivative(v)));
    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(" + "));
  }
}
```

Example 1.31. Let's clarify the distinction between the `Comparable` and `Comparator` interfaces. Consider an `Employee` class, which stores an employee's first and last name, as well as their salary. Suppose we want to be able to compare `Employee` instances. One option to do so is to declare `Employee` to implement the `Comparable<Employee>` interface. Therefore, `Employee` must override the `public int compareTo(Employee e)` method. Further suppose that our method will return the result of lexicographically comparing the employee's last name.

```
class Employee implements Comparable<Employee> {

  private double salary;
  private String firstName;
  private String lastName;

  Employee(String firstName, String lastName, double salary) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.salary = salary;
  }

  @Override
  public int compareTo(Employee o) {
    return this.lastName.compareTo(o.lastName);
  }

  // Getters and setters omitted.
}
```

Now, if we want to create a list of employees and sort them, we can use the static `sort` method from the `Collections` class.

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

import java.util.ArrayList;
import java.util.Collections;
import java.util.Collectors;
import java.util.List;

class EmployeeTester {

  @Test
  void testEmployeeComparable() {
    List<Employee> loe1 = new ArrayList<>();
    loe1.add(new Employee("John", "Doe", 100000));
    loe1.add(new Employee("Alex", "Smith", 120000));
    loe1.add(new Employee("Barbara", "Jones", 140000));
    loe1.add(new Employee("Cliff", "Brown", 160000));
    loe1.add(new Employee("Jane", "Davis", 180000));
    loe1.add(new Employee("Trevor", "Wilson", 200000));
    loe1.add(new Employee("Peter", "Taylor", 220000));
    loe1.add(new Employee("Jennifer", "Clark", 240000));
    Collections.sort(loe1);
    assertEquals("Brown, Clark, Davis, Doe, Jones, Smith, Taylor, Wilson",
                 loe1.stream()
                     .map(e -> e.getLastName())
                     .collect(Collectors.join(", ")));
  }
}
```

By default, `Collections.sort` will sort the provided collection using the object's `compareTo` method. So, in this case, the employees are sorted based on their last name. Note that `Collections.sort` uses an in-place sorting algorithm, which means that the original list is modified.

Now, suppose that we want to compare employees using a *different* metric. For instance, what if we want to sort the employees based on their first name, or perhaps their salary? One approach would be to change how `compareTo` is implemented in the `Employee` class. The problem with this is that any code that relies on the last name ordering is now broken. Plus, it's possible that the source code of the `Employee` class is immutable, a commonality when working with third-party libraries or legacy codebases.

A solution to this predicament is to use a `Comparator` object. Comparators, as their name suggests, compare instances of a class. The essential difference between a `Comparator` and `Comparable` is that the class of interest, e.g., `Employee`, should *not* implement `Comparator`. Rather, we want to create a separate class that represents a comparison between `Employee` objects by an arbitrary metric. For example, in the following listing is a class `EmployeeFirstNameComparator`, which compares employees by their first name. Notice that the `Comparator` class provides the `public int compare(T o1, T o2)` method instead of `compareTo`. Another difference is that `compare` receives two arguments rather than one, because `compareTo` relies on `this` and its other argument to perform the relevant comparison. That is, `compareTo` returns a comparison result based on `this` and its argument. By contrast, `compare` receives *two* instances of the class of interest, and the returned value should be the result of however we choose to compare those two objects.

We can then pass an instance of this comparator as a second argument to the `Collections.sort` method:

```
import java.util.Comparator;

class EmployeeFirstNameComparator implements Comparator<Employee> {

  @Override
  public int compare(Employee o1, Employee o2) {
    return o1.getFirstName().compareTo(o2.getFirstName());
  }
}
```

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

import java.util.ArrayList;
import java.util.Collections;
import java.util.Collectors;
import java.util.List;

class EmployeeTester {

  @Test
  void testEmployeeComparable() {
    List<Employee> loe1 = new ArrayList<>();
    // ... assume the same list as before.
    Collections.sort(loe1, new EmployeeFirstNameComparator());
    assertEquals("Alex, Barbara, Cliff, Jane, Jennifer, John, Peter, Trevor",
                 loe1.stream()
                     .map(e -> e.getFirstName())
                     .collect(Collectors.joining(", ")));
  }
}
```

Let's create another comparator for comparing employees based on their salaries. A lower salary indicates a lower index in the ordering relation.

```
import java.util.Comparator;

class EmployeeSalaryComparator implements Comparator<Employee> {

  @Override
  public int compare(Employee o1, Employee o2) {
    return (int) Math.signum(o1.getSalary() - o2.getSalary());
  }
}
```

1.4 Inheritance

When we introduced interfaces, we stated that they group classes that enact similar behaviors. *Inheritance* describes an "IS-A" relationship between two classes. That is, one class C_1 "is" another class if it extends the C_2 class. In the example, C_1 is called the *subclass*, and C_2 is called the *superclass*. A subclass inherits all the non-private methods and fields from its superclass. Classes can only extend one class at a time, unlike other programming languages such as C++ (Stroustrup, 2013).

In Java, every class has an implicit superclass: `Object`, which introduced the paradigm that "everything is an object in Java." The `Object` class serves as a barebones "template," of sorts, that provides the essentials for a class definition. These essentials include methods for comparing one object against another via `equals`, computing the hash code via `hashCode`, and stringifying the class via `toString`. We have seen these three methods before in a variety of contexts, but we now elaborate on their origins.

Example 1.32. When inheriting from a class, as we described, all non-private properties are inherited. So, let's consider an example that we have seen before: the `Point` class. As we recall, it stores x and y coordinates. Though, what if we want to store a "color" value inside the `Point` class? Does it make sense to modify the implementation of `Point` to now include a color? Absolutely not, because any existing code that makes use of `Point` presumes only knowledge of two coordinate values and not a color. Consequently, we should *extend* the `Point` class in a new subclass called `ColorPoint`. Importantly, do **not** copy any code from the `Point` class into `ColorPoint`, because that defeats the purpose of class extension/inheritance. Our `ColorPoint` class constructor will call the superclass constructor, via `super`, to pass the provided x and y coordinates up to the superclass definition. Remember that `x` and `y` have private access inside `Point`, meaning `ColorPoint` cannot initialize their values directly.

In designing the `ColorPoint` class, we will override the superclass implementation of `equals`, `hashCode`, and `toString` to also include the color of the point. What's convenient is that we do not need to repeat the existing comparison, hash code calculation, concatenation code respectively. Instead, we call the superclass variant of the method via '`super.equals`' or '`super.toString`'. Two `ColorPoint` instances are equal according to `equals` if their colors are the same and their coordinate values are equal. What we mean by "we do not need to repeat the existing comparison," is that the `equals` method inside `ColorPoint` should not (and will not) compare its x and y coordinates to those of the parameter.

Identical to interfaces, we should initialize an instance as its superclass, but instantiate it as a subclass type.

```
class ColorPointTester {

  @Test
  void testColorPoint() {
    Point p1 = new ColorPoint(3, 4, "RED");
    Point p2 = new ColorPoint(4, 3, "GREEN");
    Point p3 = new ColorPoint(3, 4, "RED");
    assertAll(
      () -> assertEquals("Color=RED, [x=3, y=4]", p1.toString()),
      () -> assertEquals("Color=GREEN, [x=4, y=3]", p2.toString()),
      () -> assertEquals(p1, p3),
      () -> assertNotEquals(p1, p2));
  }
}
```

```
import java.util.Objects;

class ColorPoint {

  private final String COLOR;

  ColorPoint(int x, int y, String color) {
    super(x, y);
    this.COLOR = color;
  }

  @Override
  public boolean equals(Object o) {
    ColorPoint pt = (ColorPoint) o;
    return this.COLOR.equals(pt.COLOR) && super.equals(pt);
  }

  @Override
  public int hashCode() {
    return Objects.hash(this.COLOR) + super.hashCode();
  }

  @Override
  public String toString() {
    return String.format("Color=%s, %s", this.COLOR, super.toString());
  }

  public Color getColor() {
    return this.COLOR;
  }
}
```

Some readers may question the need for inheritance; after all, could `Point` not be an interface and have `ColorPoint` implement said interface? The answer is no, because `Point` contains fields and private methods, neither of which are possible with an interface.

Example 1.33. 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.[25]

```
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()),
      () -> 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()));
  }
}
```

```
class Alien {

  enum Direction { NORTH, SOUTH, EAST, WEST };

  private int x;
  private int y;

  private Direction dir;

  Alien() {
    this.x = 0;
    this.y = 0;
    this.dir = Direction.EAST;
  }

  /**
   * Moves the alien forward by one unit in the direction it is facing.
   */
  void moveForward() {
    switch (this.dir) {
      NORTH -> this.y++;
      SOUTH -> this.y--;
      EAST -> this.x++;
      WEST -> this.x--;
    }
  }
```

[25] We base this example off of Karel J. Robot from Bergin et al. (2013) and Pattis (1995).

```
  /**
   * Turns the alien left by 90 degrees.
   */
  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.
}
```

We defined 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 must turn left three times to mimic the behavior of turning right once. One solution to this problem is to write the `turnRight` method directly inside `Alien`. Though, consider a situation where the code for `Alien`, or any arbitrary class, is hidden or immutable. In such circumstances, any extendability of functionality must come via another means.

Let's 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 detail is that we invoke the superclass constructor without parameters, because the superclass (namely `Alien`) has no constructor that receives parameters. We invoke the superclass constructor to ensure that the x, y, and direction fields are initialized.[26]

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

class RightAlienTester {

  @Test
  void testMoverAlien() {
    Alien r1 = new RightAlien();
    assertAll(
      () -> r1.turnRight(),
      () -> assertEquals(RightAlien.Direction.SOUTH, r1.getDir()),
      () -> r1.turnLeft(),
      () -> assertEquals(RightAlien.Direction.SOUTH, r1.getDir()));
  }
}
```

```
class RightAlien extends Alien {

  RightAlien() { super(); }

  /**
   * Turns the Alien right by 90 degrees.
   */
  void turnRight() {
    turnLeft();
    turnLeft();
    turnLeft();
  }
}
```

[26] Not invoking the superclass constructor may result in a `NullPointerException` when trying to access the superclass fields, because those fields were never initialized nor instantiated.

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' `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![27] One could make the argument and say that this is, in fact, a form of recursion, and while this is not incorrect, it is "nonsensical recursion" because the outcome is not only undesired, but it also never terminates.

```
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()));
  }
}
```

```
class MileMoverAlien extends Alien {

  MileMoverAlien() {
    super();
  }

  @Override
  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.

[27] Omitting '`this.`' still causes the method to infinitely loop, since not having the qualifier will cause Java to look in the current class definition.

```
import java.util.Random;

class BouncerAlien extends Alien {

  private final Random RNG;

  BouncerAlien() {
    super();
    this.RNG = new Random();
  }

  @Override
  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 a `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 an additional responsibility that may be overlooked. 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.

```
class WorldObject {

  WorldObject() {}
}
```

```
class StarObject extends WorldObject {

  StarObject() { super(); }
}
```

```
import java.lang.Class;
import java.util.ArrayList;
import java.util.List;

class WorldPosition {

  private List<WorldObject> WORLD;

  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.
   */
  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 number of rows and columns in the world. 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 a given position.

In Chapter **??**, we revisit reflection and explore its potential in greater detail. Remember that reflection is a runtime mechanism and, consequently, program performance may be penalized in certain situations.

```
final class World {

  private final WorldPosition[][] WORLD;

  World(int numRows, int numCols) {
    this.WORLD = new WorldPosition[numRows][numCols];
    for (int i = 0; i < numRows; i++) {
      for (int j = 0; j < numCols; 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.
   */
  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.
   * @param x - the x-coordinate of the position.
   * @param y - the y-coordinate of the position.
   * @return the number of stars.
   */
  int countStars(int x, int y) {
    return this.WORLD[x][y].count(StarObject.class);
  }
}
```

Example 1.34. Let's design a "role hierarchy" system for business users. In this system, there are employees that have different roles. For instance, we have managers and developers. An employee may be promotable if it implements the `IPromotable` interface. First, let's consider the hierarchy and what properties each of these roles have to offer.

An `Employee` contains a name and a unique identifier. Employees can be either hourly or salaried, meaning they receive an hourly payrate or a yearly salary. Let's categorize these into `HourlyEmployee` and `SalaryEmployee`, where both extend the `Employee` class. Accordingly, the `HourlyEmployee` class stores an hourly rate, whereas the `SalaryEmployee` class stores an annual salary amount.

A `Manager` is an `Employee` (either salaried or hourly), and contains a list of `Employee` objects that they supervise.

A `Developer` writes code in a programming language defined as a string instance variable. They also store an integer denoting their number of years of experience. Developers are strictly salaried employees. Also, a developer can be either "junior" or "senior," and a junior developer is promotable to a senior developer after they have at least five years of experience.

Let's start by designing the `Employee` class. In addition to its properties, employees can `String work()` and must override the `toString` method. The `work` method is *polymorphic*, meaning that an `Salesperson` may "work" differently compared to a `JuniorDeveloper`, but they both "work," in essence. We will say that an `Employee` works by returning a string with

their name and " is working." appended. The unique identifier that an employee has is generated by a statically-incremented counter variable, similar to how we counted instances of the Point class in Chapter 1.

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

class EmployeeTester {

  @Test
  void testEmployee() {
    Employee e1 = new Employee("Chaitrali");
    Employee e2 = new Employee("Owen");
    assertAll(
      () -> assertEquals("Chaitrali is working.", e1.work()),
      () -> assertEquals("Owen is working.", e2.work()));
  }
}
```

```
class Employee {

  private static int empCounter = 1;

  private String name;
  private int id;

  Employee(String name) {
    this.name = name;
    this.id = empCounter;
    Employee.empCounter++;
  }

  String work() {
    return this.name + " is working.";
  }

  @Override
  public String toString() {
    return String.format("Name=%s, ID=%d", this.name, this.id);
  }
}
```

The HourlyEmployee and SalaryEmployee classes, as aforementioned, extend the Employee class, the only difference being that they also receive a hourly rate and yearly salary value respectively.

To keep the conversation interesting, we refrain from overriding the work method, because there is no significant difference between how a generic Employee works and one of its direct subclasses work. By not overriding the implementation of a method, Java defaults to the existing implementation in a superclass. The toString method, on the other hand, is updated to contain the hourly rate or salary, depending on the subclass.

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

class EmployeeTester {

  @Test
  void testHourlySalaryEmployee() {
    Employee es1 = new SalaryEmployee("Andrew", 67500);
    Employee eh1 = new HourlyEmployee("Priyanka", 42.80);
    assertAll(
      () -> assertEquals("Andrew is working.", es1.work()),
      () -> assertEquals("Priyanka is working.", eh1.work()),
      () -> assertEquals("Name=Andrew, ID=1, Salary=$67500.00", es1.toString()),
      () -> assertEquals("Name=Priyanka, ID=2, Hourly=$42.80", eh1.toString()));
  }
}
```

```
class HourlyEmployee extends Employee {

  private double hrRate;

  HourlyEmployee(String name, double hrRate) {
    super(name);
    this.hrRate = hrRate;
  }

  @Override
  public String toString() {
    return String.format("%s, Hourly=%.2f", super.toString(), this.hrRate);
  }
}
```

```
class SalaryEmployee extends Employee {

  private double annualSalary;

  HourlyEmployee(String name, double annualSalary) {
    super(name);
    this.annualSalary = annualSalary;
  }

  @Override
  public String toString() {
    return String.format("%s, Salary=%.2f", super.toString(), this.annualSalary);
  }
}
```

Up next we have the `Developer` class, who programs in a language, and has two supporting subclasses: `JuniorDeveloper` and `SeniorDeveloper`. The difference between these two subclasses is their years of experience and whether they are a mentee or a mentor. Junior developers have exactly one mentor, and senior developers can have many mentees, those of which are junior developers. Because a `SeniorDeveloper` cannot be promoted, only the `JuniorDeveloper` class will implement the `IPromotable` interface. Promotion from junior to senior developer also comes with a 25% raise in salary.

Upon instantiating a JuniorDeveloper, it must receive the SeniorDeveloper who is their mentor. Inside the SeniorDeveloper class, we will expose a method that adds a JuniorDeveloper to their list of mentees.[28]

Lastly we must account for how SeniorDevelopers and JuniorDevelopers "work." A senior developer works by mentoring their mentees. A junior developer works by writing in their programming language, mentored by whomever.

```
import java.util.Set;

class EmployeeTester {

  @Test
  void testDeveloper() {
    Developer sd1 = new SeniorDeveloper("Ron", 89500, "C", 10);
    Developer d1 = new JuniorDeveloper("Calvin", 55000, "C", 1, sd1);
    Developer d2 = new JuniorDeveloper("Kushagra", 61000, "Java", 6, sd1);
    Developer d3 = new JuniorDeveloper("Tim", 57000, "C++", 2, sd1);
    assertAll(
      () -> assertEquals(Set.of(d1, d2, d3),
                         sd1.getMentees()),
      () -> assertEquals("Ron is a senior developer mentoring Calvin,
                          Kushagra, Tim.", sd1.work()),
      () -> assertEquals("Calvin is a junior developer working in C,
                          mentored by Ron.", d1.work()),
      () -> assertEquals("Kushagra is a junior developer working in
                          Java, mentored by Ron.", d2.work()),
      () -> assertEquals("Tim is a junior developer working in Java,
                          mentored by Ron.", d3.work()),
      () -> assertEquals("Name=Ron, ID=1, Salary=$89500.00, Senior
                          Developer in C with 10yoe", sd1.toString()),
      () -> assertEquals("Name=Calvin, ID=2, Salary=$55000.00, Junior
                          Developer in C with 1 yoe.", d1.toString()),
      () -> assertEquals("Name=Kushagra, ID=3, Salary=$61000.00, Junior
                          Developer in Java with 6yoe.", d3.toString()),
      () -> assertEquals("Name=Calvin, ID=4, Salary=$57000.00, Junior
                          Developer in C++ with 2yoe", d3.toString()));
  }

  @Test
  void testJuniorDeveloperPromotion() {
    Developer d1 = new JuniorDeveloper("Cole", 75000, "Python", 6);
    Developer d = new JuniorDeveloper("Adam", 45000, "Python", 0);
    assertAll(
      () -> assertTrue(d1.promote() instanceof SeniorDeveloper),
      () -> assertFalse(d1.promote() instanceof JuniorDeveloper));
  }
}
```

```
interface IPromotable {

  Employee promote();
}
```

[28] If this were a more robust and realistic system, we may choose to override equals and hashCode to take advantage of set lookups.

```
class Developer extends SalaryEmployee {

  private String language;
  private int yearsOfExperience;

  Developer(String name, double salary, String language, int yoe) {
    super(name, salary);
    this.language = language;
    this.yearsOfExperience = yoe;
  }
}
```

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

class SeniorDeveloper extends Developer {

  private List<JuniorDeveloper> mentees;

  Developer(String name, double salary, String language, int yoe) {
    super(name, salary, language, yoe);
    this.mentees = new ArrayList<>();
  }

  void addMentee(JuniorDeveloper jd) {
    this.mentees.add(jd);
  }

  @Override
  String work() {
    String names = mentees.stream()
                          .map(m -> m.getName())
                          .collect(Collectors.join(", "));
    return String.format("%s is a senior developer mentoring %s.",
                         this.getName(),
                         names);
  }

  @Override
  public String toString() {
    return String.format("%s,
                         Senior Developer in %s with %d yoe",
                         super.toString(),
                         super.getLanguage(),
                         super.getYearsOfExperience());
  }
}
```

```
class JuniorDeveloper extends Developer implements IPromotable {

  private static final double RAISE_FACTOR = 1.25;
  private static final int PROMOTION_YEARS = 5;

  private SeniorDeveloper mentor;

  Developer(String name, double salary, String lang, int yoe, SeniorDeveloper mentor) {
    super(name, salary, lang, yoe);
    this.mentor = mentor;
    this.mentor.addMentee(this);
  }

  @Override
  public Employee promote() {
    if (super.getYearsOfExperience() >= PROMOTION_YEARS) {
      return new SeniorDeveloper(super.getName(),
                                 super.getSalary() * RAISE_FACTOR,
                                 super.getLanguage(),
                                 super.getYearsOfExperience());
    } else {
      return this;
    }
  }

  @Override
  String work() {
    return String.format("%s is a junior developer working in %s,
                          mentored by %s.",
                         super.getName(),
                         super.getLanguage(),
                         this.mentor.getName());
  }

  @Override
  public String toString() {
    return String.format("%s, Junior Developer in %s with %dyoe",
                          super.toString(),
                          super.getLanguage(),
                          super.getYearsOfExperience());
  }
}
```

The developer series of classes were certainly a lot to write and design. The last class, namely `Manager`, is an hourly employee who supervises any kind of employee, including other managers. Their "work" is that they are "supervising" employee names, sorted alphabetically. The only part of this that is more complex than the others is the comparator that we provide to the `TreeSet`. Because the set stores employee instances, our comparator must receive two employees, but return a comparison based on their name. Let's see what this looks like:

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

class EmployeeTester {

  @Test
  void testManager() {
    HourlyEmployee m1 = new Manager("Abby", 30.00);
    m1.addDirectReport(new JuniorDeveloper("Cole", 75000, "Python", 6));
    m1.addDirectReport(new JuniorDeveloper("Adam", 45000, "Python", 0));
    m1.addDirectReport(new SalaryEmployee("Pete", 46000));
    assertAll(
      () -> assertEquals("Abby manages Adam, Cole, Pete.",
                         m1.work()),
      () -> assertEquals("Abby, ID=1, Hourly=30.00, Manager",
                         m1.toString()));
  }
}
```

```
import java.util.Collectors;
import java.util.Comparator;
import java.util.Set;
import java.util.TreeSet;

class Manager extends HourlyEmployee {

  private Set<Employee> EMPS;

  Manager(String name, double hrRate) {
    super(name, hrRate);
    this.EMPS = new TreeSet(employeeNameComparator());
  }

  void addDirectReport(Employee e) {
    this.DIRECT_REPORTS.add(e);
  }

  @Override
  String work() {
    return String.format("%s supervises %s",
                         super.getName(),
                         this.EMPS.stream()
                                  .map(x -> x.getName())
                                  .collect(Collectors.joining(", ")));
  }

  @Override
  public String toString() {
    return String.format("%s, Manager", super.toString());
  }

  private static Comparator<Employee> employeeNameComparator() {
    return new Comparator<Employee>() {
      @Override
      public int compare(Employee e1, Employee e2) {
        return e1.getName().compareTo(e2.getName());
      }
    };
  }
}
```

Example 1.35. Let's design a class hierarchy for a series of Mythicritters objects. In the exotic land of Mythicritters, there are three kinds of creatures: `Mage`, `Warrior`, and `Rogue`. Beneath these, each creature can transform into a more powerful form: `Archmage`, `Berserker`, and `Assassin`.

Mythicritters have a set number of health points, statistics about their power/strength-/speed, and two kinds of "attacks." Additionally, they have a level to indicate how strong they are. We will come back to the notion of an attack later. Finally, they also have either one or two types, which are returned via the `getTypes` method as a `Set<IType>`.[29] Let's design the `Mythicritter` class. Aside from the accessors and mutators, we will add one instance method for adding attacks to the `Mythicritter`, as long as there are less than two.

```
import java.util.HashSet;
import java.util.Set;

class Mythicritter {

  private int maxHealth;
  private int currentHealth;
  private int level;
  private int power;
  private Set<IAttack> attacks;

  Mythicritter(int maxHealth, int level, int power) {
    this.maxHealth = maxHealth;
    this.currentHealth = this.maxHealth;
    this.level = level;
    this.power = power;
    this.attacks = new HashSet<>();
  }

  void addAttack(Attack a) {
    if (this.attacks.size() < 4) {
      this.attacks.add(a);
    }
  }

  Set<IType> getTypes() {
    return Set.of();
  }
}
```

As we stated, Mythicritter have attacks, or ways to combat an opponent. Attack have a type, a base power statistic, and a number representing how many "uses" that the attack has remaining. Each time an attack is used, its usage counter is decremented. Once it reaches zero, the attack can no longer be used. Attacks also inhabit exactly one type, which cannot change.

[29] We will design `IType` in a few paragraphs.

```
class Attack {

  private final IType TYPE;
  private final int NUM_USES;
  private int remainingUses;
  private int basePower;

  Attack(String name, int numUses, int basePower, IType type) {
    this.name = name;
    this.baseDamage = baseDamage;
    this.NUM_USES = numUses;
    this.TYPE = type;
  }

  // Accessors and mutators omitted.
}
```

Finally, Mythicritters, as well as attacks, have associated types. Types in the world of Mythicritters are a property thereof, which affect the power of their attacks, as well as what types they are vulnerable to/strong against. All in all, it is akin to a game of "Rock, paper, scissors." That is, scissors beats paper, paper beats rock, and rock beats scissors. Mythicritters and attack types work similarly: if an attack is `ZapType` and the Mythicritter it is used against is `OceanicType`, the attack is "enhanced" against the Mythicritter. The defending Mythicritter thus takes more damage than it would otherwise. Conversely, if an attack is `InfernoType` and the Mythicritter is `BoulderType`, the attack is "vulnerable" against the Mythicritter. The defending Mythicritter thus takes less damage than it would otherwise.

Now, let's design a "base interface" that other classes implement. An `IType` contains two methods: `Set<IType> vulnerableTo()` and `Set<IType> strongAgainst()`, which returns sets to represent the types that a type is vulnerable to or strong against. These methods will call upon a data structure that we will denote as a "type registry," which defines the relationships from one type to another. Interestingly, this means that the methods inside `IType` should have a body to reference the type registry, but in the previous section we stated that the methods of an interface must not have bodies. *Default methods* of an interface, on the contrary, may have bodies, and serve as a "default implementation" of a method in the event that an interface does not override its definition.

```
import java.util.Set;

interface IType {

  default Set<IType> vulnerableTo() {
    return TypeRegistry.vulnerableTo(this);
  }

  default Set<IType> strongAgainst() {
    return TypeRegistry.strongAgainst(this);
  }
}
```

Out of conciseness, let's design only three types: `ZapType`, `OceanicType`, and `FlameType`. Repeating the rock-paper-scissors analogy, zap types are strong against oceanic type but weak to flame type. Oceanic types are strong against flame type but weak to zap. Flame types are strong against zap but weak to oceanic.

These three types, namely zap, oceanic, and flame, are designed as classes that exist solely for the purposes of instantiating exactly one instance thereof. Types do not need to be in-

stantiated multiple times since they are immutable and do not contain any relevant state. We are taking advantage of a *design paradigm* called *singleton*. The class constructors are private to prevent outside instantiation of the types, but the sole instance is public, static, and final for global, unrestricted, and immutable access.

```
class FlameType implements IType {

  static final IType FLAME_TYPE = new FlameType();

  private FlameType() {}
}
```

```
class OceanicType implements IType {

  static final IType OCEANIC_TYPE = new OceanicType();

  private OceanicType() {}
}
```

```
class ZapType implements IType {

  static final IType ZAP_TYPE = new ZapType();

  private ZapType() {}
}
```

The aforementioned type registry is a class that stores two maps of type weaknesses and strengths. It provides two static methods: `vulnerableTo` and `strongAgainst`, identical to the `IType` interface, which tap into the map data structures for querying.

```
import java.util.Map;
import java.util.Set;

class TypeRegistry {

  private static final Map<IType, Set<IType>> vulnerabilities =
    Map.of(OceanicType.OCEANIC_TYPE, Set.of(ZapType.ZAP_TYPE),
           ZapType.ZAP_TYPE, Set.of(FlameType.FLAME_TYPE),
           FlameType.FLAME_TYPE, Set.of(OceanicType.OCEANIC_TYPE));

  private static final Map<IType, Set<IType>> strengths =
    Map.of(OceanicType.OCEANIC_TYPE, Set.of(FlameType.FLAME_TYPE),
           ZapType.ZAP_TYPE, Set.of(OceanicType.OCEANIC_TYPE),
           FlameType.FLAME_TYPE, Set.of(ZapType.ZAP_TYPE));

  private TypeRegistry() {};

  static Set<IType> vulnerableTo(IType t) {
    return vulnerabilities.getOrDefault(t, Set.of());
  }

  static Set<IType> strongAgainst(IType t) {
    return strengths.getOrDefault(t, Set.of());
  }
}
```

Inside the `Mythicritter` class, let's design two methods: `vulnerableTo` and `strongAgainst`, both of which invoke the type registry. The problem, though, is that we potentially

need to take the union of two sets, since a `Mythicritter` stores a set of types. To compensate, we can use streams to take the union of multiple sets, which collapses the problem from handling both one and two sets independently into just handling a stream. To combine the sets, we can use the `flatMap` stream method, which is a higher-order method to apply a lambda expression to a list of collections, then flatten the result into a single collection. Namely, we take the sets of types, map the `vulnerableTo` or `strongAgainst` over those types, creating a stream of sets of types. These are then converted into streams themselves and then flattened into a single stream. Finally, we collect those results into a set.

Before we write these two methods, however, let's actually create a `Mythicritter` instance to show off our hard work. Let's design the `Mage` class, which is a zap type `Mythicritter`, hence the subclass/superclass relationship. We will instantiate it to start off with 50 health, level 1, and have a base "power" statistic of 4. Because we must also pass the type information to the superclass constructor, we need to invoke `super(...)` with information provided from the subclass contructor in addition to the type(s). After creating the subclass instance, we can write tests for the type registry.

```
import java.util.Set;

class Mage extends Mythicritter {

  Mage(int health, int level, int power) {
    super(health, level, power);
  }

  @Override
  Set<IType> getTypes() {
    return Set.of(ZapType.ZAP_TYPE);
  }
}
```

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

import java.util.Set;

class MythicritterTester {

  @Test
  void testMythicritter() {
    Mythicritter p1 = new Mage(50, 1, 4);
    assertAll(
      () -> assertEquals(Set.of(FlameType.FLAME_TYPE),
                         p1.weakTo()),
      () -> assertEquals(Set.of(OceanicType.OCEANIC_TYPE),
                         p1.strongAgainst()));
  }
}
```

Finally, let's represent an important component of Mythicritters: transformations! A Mythicritter can transform from one form to another. From this, a Mythicritter's type can change, as do its base statistics and health. For example, the transformation of Mage into Archmage adds the `MythicalType` to its type set. We can change the type that a `Mythicritter` inhabits by overriding the `getTypes` method: we retrieve the superclass type set, then add the new type earned by transforming, should it exist.

To make transformation a bit more interesting, we will design the `ITransformable` interface, which is implemented by any transformable Mythicritter. This interface contains two

methods: `boolean canTransform()` and `Mythicritter transform()`. The former describes the requirements before a Mythicritter can transform, e.g., whether or not it has to be at a certain level. The latter `transform` method returns a new instance of the transformation if it can transform, and itself otherwise.

Designing an interface in this way means that Mythicritters that transform once, but cannot transform thereafter must also override the methods in the `ITransformable` interface to return `false` and themselves respectively.

For contextualization, Mages transform into Archmages. A Mage can transform into an Archmage once it reaches level 30. Because `MythicalType` is almost a carbon copy of the other three types, we omit its inclusion.

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

class MythicritterTester {

  @Test
  void testTransformation() {
    Mythicritter p1 = new Mage(20, 1, 4);
    Mythicritter p2 = new Mage(1000, 40, 100);
    assertAll(
      () -> assertFalse(p1.canTransform()),
      () -> assertTrue(p1.transform() instanceof Mage),
      () -> assertTrue(p2.canTransform()),
      () -> assertTrue(p2.transform() instanceof Archmage));
  }
}
```

```
interface ITransformable {

  boolean canTransform();

  Mythicritter transform();
}
```

```
class Mage extends Mythicritter implements ITransformable {
  // ... previous methods not shown.

  @Override
  public boolean canTransform() {
    return this.getLevel() >= 30;
  }

  @Override
  public Mythicritter transform() {
    if (!this.canTransform()) {
      return this;
    } else {
      return new Archmage(super.getHp() * 2, super.getLevel(), super.getPower() * 3);
    }
  }
}
```

```
import java.util.Collectors;
import java.util.Set;
import java.util.Stream;

class Archmage extends Mage implements ITransformable {

  Archmage(int health, int level, int power) {
    super(health, level, power);
  }

  @Override
  public boolean canTransform() {
    return false;
  }

  @Override
  public Mythicritter transform() {
    return this;
  }

  @Override
  Set<Type> getTypes() {
    Stream<IType> ot = super.getTypes().stream();
    Stream<IType> nt = Set.of(MythicalTypeType.MYTHICAL_TYPE).stream();
    return Stream.concat(ot, nt)
                 .collect(Collectors.toSet());
  }
}
```

We went through all of this trouble to create a complex system, so what is its intended purpose? With a bit of work up front, we made it easy to extend this system to include new types, new kinds of Mythicritters, and much more.[30]

1.5 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 1.36. 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`.

[30] We did not do anything with the attacks of a Mythicritter; this may be a good place to start expanding upon if you are interested!

```
abstract class Animal {

  Animal() {}

  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!"`.

```
class Dog extends Animal {

  Dog() { super(); }

  @Override
  String speak() { return "Woof!"; }
}
```

```
class Cat extends Animal {

  Cat() { super(); }

  @Override
  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.[31]

Example 1.37. 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.

[31] Both abstract classes and interfaces can contain static methods and variables.

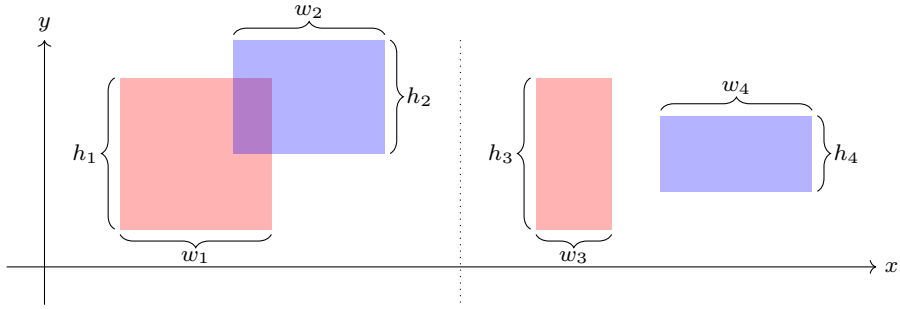

Fig. 1.2: Collision Detection Between Rectangles.

```
abstract class GameObject {

  private int x;
  private int y;

  GameObject(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
```

```
abstract class AxisAlignedBoundingBoxObject extends GameObject {

  private int width;
  private int height;

  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.
   */
  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);
  }
}
```

We declared an abstract class to extend another abstract class; which is perfectly acceptable. Because it makes no sense to instantiate an entity, in and of itself, 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 `AxisAligned-`

BoundingBox 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.

```
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 is certainly different! Let's, for the sake of emphasizing the distinctions, design a IDrawable interface, which provides one method: void 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.

```
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);
}
```

```
class CircleObject extends AxisAlignedBoundingBoxObject implements IDrawable {

  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());
  }
}
```

```
class RectangleObject extends AxisAlignedBoundingBoxObject implements IDrawable {

  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 1.38. 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`.

```
import java.util.List;
import java.util.Map;

abstract class Argument {

  private String key;

  Argument(String key) { this.key = key; }

  String getKey() { return this.key; }

  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.

```
import java.util.List;
import java.util.Map;

class OptionalArgument extends Argument {

  OptionalArgument(String key) {
    super(key);
  }

  @Override
  boolean validate(Map<String, List<String>> args) {
    return true;
  }
}
```

```
import java.util.List;
import java.util.Map;

class NumberedArgument extends Argument {

  private final int NUM_REQUIRED_ARGS;

  NumberedArgument(String key, int n) {
    super(key);
    this.NUM_REQUIRED_ARGS = n;
  }

  @Override
  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.

```
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.HashSet;

class ArgumentParser {

  private final Map<String, List<String>> PARSED_ARGS;
  private final Set<Argument> ARGS;

  ArgumentParser(String[] args) {
    this.ARGS = new HashSet<>();
    this.PARSED_ARGS = new HashMap<>();
    String currKey = null;
    for (String arg : this.ARGS) {
      if (arg.startsWith("--")) {
        currKey = arg.split("--")[1];
        this.PARSED_ARGS.putIfAbsent(currKey, new ArrayList<>());
      } else if (currKey != null) {
        this.PARSED_ARGS.get(currKey).add(arg);
      }
    }
  }

  void addArgument(Argument arg) {
    this.ARGS.add(arg);
  }

  List<String> getArguments(String key) {
    if (this.PARSED_ARGS.containsKey(key)) {
      return this.PARSED_ARGS.get(key);
    } else {
      return 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.

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

class ArgumentParser {

  /**
   * Determines whether or not all of the arguments in the stored
   * instance variable are "valid".
   * @return true if all arguments are valid, false otherwise.
   */
  boolean parseArguments() {
    return this.arguments.stream()
                         .allMatch(e -> e.validate(this.parsedArguments));
  }
}
```

The ASPL Interpreter

Example 1.39. 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 called ASPL, or "A Simple 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.

As a means of motivation, let's write a few programs in this language to show its capabilities. The first listing is a simple program to add two numbers together. The second listing binds two variables, and if their sum is equal to 42, then the result is 100, otherwise it is zero. The third listing declares a variable, followed by a conditional, both cases of which contain another binding of a variable, closing off with a product operation.

```
(+ 25 17)
```

```
(let ([x 10])
  (let ([y 32])
    (if (eq? (+ x y) 42)
        100
        0)))
```

```
(let ([z 95])
  (if (eq? (- 100 5) z)
      (let ([w -10])
        (* w z 2))
      (let ([w -5])
        (* z w -2))))
```

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.

Each abstract syntax tree should be evaluable as a means of reducing the expression to its simplest form. For example, numbers and booleans, the literals of our language, resolve to themselves. Primitive operations apply the operation to its arguments, then ultimately reduce to a value. Conditionals resolve to either of its branches, and "let" bindings resolve to its body. We will cover each case one-by-one as we cover the material. For now, though, the abstract syntax tree node class contains the `eval` method, which denotes how a node is to be evaluated.

```
import java.util.List;

abstract class AstNode {

  private final List<AstNode> CHILDREN;

  AstNode(List<AstNode> children) {
    this.CHILDREN = children;
  }

  AstNode(AstNode... children) {
    this(List.of(children));
  }

  abstract AstNode eval();

  List<AstNode> getChildren() {
    return this.CHILDREN;
  }

  public abstract String toString();
}
```

From here, the simplest two abstract syntax tree nodes are `NumNode` and `BoolNode`, corresponding to numbers and booleans literals respectively. Nodes that encapsulate sole values, e.g., numbers and booleans, are examples of literals, and all literals are evaluated/treated the same way. As such, let's design the generic and abstract `LiteralNode<T>` class, which all literal types extend, those for our purposes being `NumNode` and `BoolNode`.[32]

A `LiteralNode<T>` stores an immutable object of type `T`. Literal values resolve to themselves and only themselves, which means that they return `this` as the object in `LiteralNode`'s `eval` method. The `LiteralNode` class overrides `equals`, `hashCode`, and `toString` to compare literals, return the hash code of the stored instance variable, and "stringify" the instance variable respectively. Designing the `LiteralNode` class in this fashion makes designing the subclasses easier and significantly less redundant, because `NumNode` and `BoolNode` need to only provide constructors that invoke the superclass.

[32] We say, "for our purposes," because the language could also support string or character literals.

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

class AstTest {

  @Test
  void testNumNode() {
    assertEquals("42", new NumNode("42").toString());
  }

  @Test
  void testBoolNode() {
    assertEquals("true", new BoolNode("true").toString());
    assertEquals("false", new BoolNode("false").toString());
  }
}
```

```
import java.util.Objects;

abstract class LiteralNode<T> extends AstNode {

  private final T VALUE;

  LiteralNode(T value) { this.VALUE = value; }

  @Override
  AstNode eval(Environment env) { return this; }

  @Override
  public boolean equals(Object o) {
    if (!(o instanceof LiteralNode)) { return false; }
    else { return this.VALUE.equals(((LiteralNode<?>) o).VALUE); }
  }

  @Override
  public int hashCode() { return Objects.hashCode(this.VALUE); }

  @Override
  public String toString() { return this.VALUE.toString(); }

  T getValue() { return this.VALUE; }
}
```

```
final class NumNode extends LiteralNode<Double> {

  NumNode(double value) { super(value); }

  NumNode(String value) { super(Double.parseDouble(value)); }
}
```

```
final class BoolNode extends LiteralNode<Boolean> {

  BoolNode(boolean value) { super(value); }

  BoolNode(String value) { this(Boolean.parseBoolean(value)); }
}
```

The next logical step is to add primitive operations via `PrimNode`. A primitive operator is an operation akin to addition, subtraction, value equality, and so forth. Primitive operators receive any number of arguments, and the behavior of which is handled as a case analysis of the `eval` method.

Though, evaluating a primitive operation is not as simple as applying the operator to its arguments. Consider the following code segment, where we have the primitive operation `+` applied to two more primitive operations '`*`' and '`-`.' We see that it is impossible to directly apply the plus operation to the two primitives, because addition only works over `NumNode` values and not `AstNode` instances. So, when applying a primitive operation, we must first recursively evaluate its children nodes, i.e., the arguments. The list of arguments is converted into a stream, where we map the `eval` method over each element. Afterwards, we write the aforementioned case analysis on the operation. For the addition operator, we sum all the values of the children nodes. Even though each node is definitionally an `AstNode`, we can safely cast them to `NumNode` because we know that the operation is semantically valid only over numbers (the same logic applies to other such primitive operators).

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

class AstTest {

  @Test
  void testPrimNode() {
    assertAll(
      () -> assertEquals(new NumNode(42),
                         new PrimNode("+",
                          new NumNode(25),
                          new NumNode(17)).eval());
      () -> assertEquals(new NumNode(42),
                         new PrimNode("-",
                          new NumNode(97),
                          new NumNode(55)).eval());
      () -> assertEquals(new NumNode(42),
                         new PrimNode("*",
                          new NumNode(6),
                          new NumNode(1),
                          new NumNode(7)).eval());
      () -> assertEquals(new BoolNode(true),
                         new PrimNode("eq?",
                          new PrimNode("+",
                           new NumNode(5),
                           new NumNode(37)),
                           new NumNode(42)).eval())):
      () -> assertEquals(new NumNode(42),
                         new PrimNode("+",
                          new PrimNode("*",
                           new NumNode(2),
                            new PrimNode("-",
                             new NumNode(57),
                             new NumNode(23))),
                           new PrimNode("-",
                            new NumNode(4),
                            new NumNode(30))));
  }
}
```

```
import java.util.List;

final class PrimNode extends AstNode {

  private final String OP;

  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
  AstNode eval(Environment env) {
    List<AstNode> operands = this.getChildren().stream()
                                               .map(n -> n.eval(env))
                                               .toList();
    switch (this.OP) {
      case "+": return this.primPlus(operands, env);
      case "-": return this.primMinus(operands, env);
      case "*": return this.primProduct(operands, env);
      case "eq?": return this.primEq(operands, env);
      default: return null;
    }
  }

  @Override
  public String toString() {
    return String.format("(%s %s)",this.OP, this.getChildren().toString());
  }

  private AstNode primPlus(List<AstNode> args, Environment env) {
    return new NumNode(args.stream()
                           .map(t -> ((NumNode) t).getValue())
                           .sum());
  }

  private AstNode primMinus(List<AstNode> args, Environment env) {
    double res = ((NumNode) args.get(0)).getValue();
    for (int i = 1; i < args.size(); i++) {
      res -= ((NumNode) args.get(i)).getValue();
    }
    return new NumNode(res);
  }

  private AstNode primProduct(List<AstNode> args, Environment env) {
    return new NumNode(args.stream()
                           .map(t -> ((NumNode) t).getValue())
                           .reduce(1.0, (a, c) -> c * a));
  }

  private AstNode primEq(List<AstNode> args, Environment env) {
    return new BoolNode(args.get(0).equals(args.get(1)));
  }
}
```

Adding the equality comparison operator provides a pathway to designing the conditional expression, namely `IfNode`. An `IfNode` contains three children nodes: a predicate, a consequent, and an alternative. The `eval` method of an `IfNode` evaluates the predicate, and if it is true, evaluates the consequent, otherwise it evaluates the alternative. The predicate *must* resolve to a boolean, assuming the program is well-formed.

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

class AstTest {

  @Test
  void testIfNode() {
    assertAll(
      () -> assertEquals(new NumNode(100),
                         new IfNode(new BoolNode(true),
                                    new NumNode(100),
                                    new NumNode(0)).eval()),
      () -> assertEquals(new BoolNode(false),
                         new IfNode(new PrimNode("eq?",
                                      new NumNode(5),
                                      new NumNode(5)),
                                    new BoolNode(true),
                                    new BoolNode(false)).eval()));
  }
}
```

With numbers, booleans, primitive operations, and conditionals taken care of, we now come to the challenging part: variable bindings. We need a way of introducing variable bindings to their values, 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 is 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>)`. In tandem, we will also write the `VarNode` class, which represents variable placeholders. In order to do anything meaningful with both classes, we need to discuss the scope of a variable and how to handle the encompassing issues.

The *scope of a variable* refers to its lifetime. Consider the following program in our language. Initially, the program has no variable bindings. After encountering the `let`, we enter a scope that binds the variable identifier x to 5. The body of this `let` is, therefore, the *scope* of x. Inside this body, we encounter yet another variable, namely y, which binds y to 10. The body of this `let` is the scope of y. Outside the scope of the body, y is non-existent.

```
(let ([x 5])
  (let ([y 10])
    (* x y)))
```

When executing this program in our heads, we know intuitively that x refers to 5 and y refers to 10. To write a programming language, though, we need to formalize the notion of "variable lookup." That is, we must define how to associate variable identifiers to values. Of course, the best data structure for value association is a map, and indeed, this is the structure we will use.

Programming languages use *environments* to associate identifiers to values. Upon encountering a variable declaration, we extend the current environment to contain the new binding. This begs the question, "Why not just modify the environment?" The answer is that we want to respect the scope of variables. Modifying the environment changes the environment for

all scopes, an undesired trait. In other words, if we mutate a variable binding in the existing environment, the variable's lifetime is extended to the entire program rather than to the scope in which it was declared. Instead, we *extend* the environment, which establishes a link between the current environment and the newly-declared environment. This way, we can look up variables in the current environment, and if they do not exist, we can look them up in the parent environment. If the variable does not exist in the parent environment, we return a `null` value.

Environments, accordingly, contain two instance variables: a `Map<String, AstNode` and a `Environment` parent pointer. Our environment class comprises two methods: `lookup` and `extend`.

The `lookup` method attempts to find a binding for the given variable identifier using the aforementioned approach. The `extend` method instantiates a new environment, whose parent is the current existing environment. This newly-instantiated environment, importantly, contains a new variable binding. Remember that the environment is a functional data structure, meaning that it is immutable.

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

class EnvironmentTester {

  @Test
  void testEnvironment() {
    Environment root = new Environment();
    Environment e1 = root.extend("x", new NumNode(5));
    Environment e2 = e1.extend("y", new NumNode(6));
    assertAll(
      () -> assertEquals(new NumNode(5), e2.lookup("x")),
      () -> assertEquals(new NumNode(6), e2.lookup("y")),
      () -> assertEquals(null, e2.lookup("z")));
  }
}
```

```
import java.util.HashMap;
import java.util.Map;

final class Environment {

  private final Map<String, AstNode> ENV;
  private final Environment PARENT;

  Environment(Environment parent) {
    this.ENV = new HashMap<>();
    this.PARENT = parent;
  }

  Environment() { this(null); }

  AstNode lookup(String id) {
    if (this.ENV.containsKey(id)) { return this.ENV.get(id); }
    else if (this.PARENT != null) { return this.PARENT.lookup(id); }
    else { return null; }
  }

  Environment extend(String id, AstNode value) {
    Environment env = new Environment(this);
    env.ENV.put(id, value);
    return env;
  }
}
```

Now we can design the `VarNode` class. The apparent question is, "How do we evaluate a variable?" Using environments, we look up the variable identifier and return its associated abstract syntax tree. But, where does the environment come from? Environments are passed as an argument to the `eval` method, which means that all previously-existing `eval` methods must be updated to accept an environment as an argument.

Variable nodes, on their own, are simple yet relatively unuseful, and simply cannot exist without a means of introducing them into the environment context. This is where the `LetNode` class comes into play. The `LetNode` class introduces a new variable binding into the environment. A `LetNode` has two abstract syntax tree children: an expression and a body. The expression is evaluated and its value is bound to the provided variable identifier in an *extended environment* e_2, whose parent is the current environment e_1. The body of the `LetNode` is evaluated with respect to the extended environment, i.e., e_2.

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

class AstTest {

  @Test
  void testLetNode() {
    Environment env = new Environment();
    assertAll(
      () -> assertEquals(new NumNode(42),
                         new LetNode("x",
                          new NumNode(42),
                          new VarNode("x")).eval(env)),
      () -> assertEquals(new NumNode(42),
                         new LetNode("x",
                          new NumNode(1),
                          new LetNode("y",
                           new NumNode(41),
                           new PrimNode("+",
                            new VarNode("x"),
                            new VarNode("y")))).eval(env)));
  }
}
```

```
final class VarNode extends AstNode {

  private final String NAME;

  VarNode(String name) {
    super();
    this.NAME = name;
  }

  String getName() {
    return this.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
  AstNode eval(Environment env) {
    String id = this.NAME;
    AstNode res = env.lookup(id);
    return res.eval(env);
  }

  @Override
  public String toString() {
    return this.NAME;
  }
}
```

```
final class LetNode extends AstNode {

  private final String ID;

  LetNode(String id, AstNode exp, AstNode body) {
    super(exp, body);
    this.ID = id;
  }

  /**
   * Interprets a let statement. The body is evaluated in the extended env.
   * The extended environment contains the binding introduced by the ID.
   * The identifier's binding expression "exp" is evaluated in "env".
   * @param env - The environment to use for the let.
   * @return The result of the let statement.
   */
  @Override
  AstNode eval(Environment env) {
    String id = this.ID;
    AstNode exp = this.getChildren().get(0);
    AstNode body = this.getChildren().get(1);

    // Interpret the expression and convert it into its AST.
    AstNode newExp = exp.eval(env);
    Environment e1 = env.extend(id, 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.ID, e, b);
  }
}
```

Finally, at long last, we can write some comprehensive tests! We will store each test in the `InterpTester` class, which polymorphically executes `eval` on the abstract syntax tree instances. All examples are initialized with an empty environment, because there are no (locally-declared) variable bindings at the start of a program. Unfortunately, we still have to

write the programs as a series of compositional abstract syntax trees, but the problems of lexing and parsing raw string input into an abstract syntax tree are reserved for another time (or perhaps a separate course altogether).

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

class InterpTester {

  @Test
  void testEval() {
    assertAll(
      () -> assertEquals(new NumNode("42"),
                         new NumNode("42").eval(new Environment())),
      () -> assertEquals(new BoolNode(true),
                         new PrimNode("eq?",
                          new NumNode(42),
                          new PrimNode("-",
                           new NumNode(100),
                           new NumNode(58))).eval(new Environment())),
      () -> assertEquals(new NumNode("42"),
                         new LetNode("x",
                          new NumNode("42"),
                          new VarNode("x")).eval(new Environment())),
      () -> assertEquals(new NumNode("42"),
                         new LetNode("x", new NumNode("1"),
                          new LetNode("y", new NumNode("41"),
                           new PrimNode("+",
                            new VarNode("x"),
                            new VarNode("y")))).eval(new Environment())));
  }
}
```

Object-oriented programs with inheritance should be structured as a sequence of specific subclasses that extend an abstract class, as we have demonstrated with the different abstract syntax tree node types, and the root `AstNode` abstract class.

1.6 Exercises

Exercise 1.1. (⋆)
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 1.2. (⋆)
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 1.3. (⋆)
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 1.4. (⋆)
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.

The instance variables, accessors, and mutators (where applicable) should be named as follows:

- `id`, `getId`.
- `firstName`, `getFirstName`.
- `lastName`, `getLastName`.
- `birthYear`, `getBirthYear`.
- `salary`, `getSalary`, `setSalary`

Exercise 1.5. (⋆)
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 1.6. (⋆⋆)
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 and a space. Do not add a comma and space after the last field.

Exercise 1.7. (⋆⋆)
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 `List<Employee>` as an instance variable. Whether you choose to instantiate it as an `ArrayList` or a `LinkedList` is up to you and makes little difference for this particular question. Its constructor should receive no arguments. The instance variable, along with its accessor and mutator, should be named `employees`, `getEmployees`, and `setEmployees` respectively.
(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 `public String toString()` method to print out the list of employees in the `Job`. To make this easy, you can simply invoke the `toString` method from the `List` implementation.

Exercise 1.8. (⋆⋆)
In this exercise you will design a simple music system, similar to Spotify.

(a) Design the `Song` class, which stores its `title`, `artist`, `genre`, and `length`. The first three fields are strings and the last is an integer. Create the accessor methods, then override `equals`, `hashCode`, and `toString`.
(b) Design the `Playlist` class, which stores the title of the playlist and a set of the songs in the list. In this class, create the accessor methods, then override `equals`, `hashCode`, and `toString`. Finally, design the `boolean addSong(Song s)` that attempts to add s to the set of songs. If it already exists, return false. Otherwise, add the song to the set and return true.
(c) Design the `User` class, which stores their name and a list of playlists. Its constructor should receive the name and assign it to the respective instance variable. Instantiate the list to a new `ArrayList`.

    (i) Design the `boolean createPlaylist(String t, Song... S)` method, which receives a playlist title t and a variadic number of songs S, attempts to create a playlist

with the title t. If it already exists, return false and do nothing else. Otherwise, declare a new `Playlist` to their list and add to it the given songs.

(ii) Design the `Playlist getPlaylist(String t)` method that, when given a playlist title t, returns the `Playlist` instance with that title. If such a playlist does not exist, return `null`.

(d) Design the `MusicSystem` class, which stores a list of users and a set of all the songs in the system. In the constructor, instantiate these to an `ArrayList` and `HashSet` respectively.

(i) Design the `void addUser(User u)` method that receives a user u and adds them to the list of users.

(ii) Design the `boolean addSong(User u, String t, Song s)` method that, when given a user u, a playlist title t, and a song s, adds s to u's playlist with the title t. If u does not exist or t is not a title in a playlist authored by u, return false.

(iii) Design the `Map<User, Song> getLongestSong()` method that returns the longest (length) song out of all the songs that a user has in their playlists. If the user has no playlists nor any songs in the playlists, do not add that user to the map.

Exercise 1.9. ($\star\star$)
In this exercise you will create two classes: `Chocolate` and `ChocolateBox` to store a two-dimensional array of chocolate pieces.

(1) Design the `Chocolate` class, which stores a string denoting its kind, and an integer representing its weight in ounces.

(2) Design the `ChocolateBox` class, which stores a `Chocolate[][]` array as an instance variable. Its constructor should receive the number of rows and columns of the box.

(i) Design the `int numberOfChocolates()` method that returns the number of non-`null` instances of `Chocolate` that are in the `ChocolateBox`.

(ii) Design the `void shuffleChocolate()` method, which randomizes the elements in the box. How you shuffle them is up to you, as long as it is a sufficient shuffle (and not just, for example, a linear shift of all chocolates).

(iii) Design the `int removeFirst(String kind)` method, which removes the first occurrence, from the top, of the `kind` of chocolate. Return what position was removed, assuming positions are numbered from 1 to n, ordered from left to right, then top to bottom (similar to a standard calendar). If there are no `kind` of chocolates in the box, return `-1`.

(iv) Design the `ChocolateBox allergyBox(String kind)` method, which returns a new `ChocolateBox` where all `kind` of chocolates are removed. If there are `null` spots in between the chocolates of the old box, shift the chocolates over accordingly. Consider the following `ChocolateBox`:

Dark	*White*	*Milk*	*Nut*	*Sweet*
`null`	*Nut*	*Dark*	`null`	`null`
White	*Sweet*	*Nut*	`null`	*Nut*

Invoking `allergyBox("Nut")` on this box would return the following `ChocolateBox`:

Dark	*White*	*Milk*	*Sweet*	*Dark*
White	*Sweet*	`null`	`null`	`null`
`null`	`null`	`null`	`null`	`null`

Exercise 1.10. ($\star\star$)
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:

$$next = |r_n \cdot 1103515245 + 12345|$$
$$r_{n+1} = \left(\frac{next}{2^{16}}\right) \% \, 2^{15};$$

(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 1.11. ($\star\star\star$)
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. Do *not* simply assign the provided array to the instance variable!
(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 `boolean 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, return false and do not add the matrix.
(d) Design the `boolean multiply(Matrix m)` method, which multiplies the values of the passed matrix to the current matrix. If we cannot multiply m with this matrix, return false and do not multiply the matrix.
(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 `public String toString()` method to return a stringified version of the matrix. As an example, `[[1, 2, 3], [4, 5, 6]]` represents the following matrix:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Exercise 1.12. (⋆⋆)
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 `public String toString()` method to call the `toString` methods of the two instance variables, conjoined by a semicolon.

Exercise 1.13. (⋆⋆)
This exercise has three parts.

(a) Design the `GameRunner` class, which stores a list of objects `List<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 1.14. (⋆⋆)
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 1.15. ($\star$)
Design the `void set(T e, int idx)` method within the `MiniArrayList` class, which sets the element at *idx* to the given *e* element.

Exercise 1.16. ($\star$)
Design the `void isEmpty()` method within the `MiniArrayList` class, which returns whether or not the list is empty.

Exercise 1.17. ($\star\star$)
Design the `void clear()` method within 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 1.18. ($\star\star$)
Override the `public boolean equals(Object o)` method 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 1.19. ($\star\star$)
Using the `StackFrame` class, design an implementation of the tail recursive factorial method.

Recall how to do this from Chapter **??**: instead of pushing an activation record to the call stack, we can simply update the bindings in the existing frame.

Exercise 1.20. (⋆⋆)
This exercise has seven parts.

In this question you will design the `Time` class for working with units of time. Programming languages often support operations for handling dates and times to varying degrees of success. Java provides a few classes and methods of its own, and you cannot use these in your implementation, as that would defeat the point of the exercise.

(a) Design the `Time` class. It should contain three constructors that receive the following parameters:

- `Time(int h, int m, int s)` receives three integers `h`, `m`, and `s` represents times in hours, minutes, and seconds respectively.
- `Time(int s)` receives a single integer `s` representing the number of seconds.
- `Time(String t)` receives a string of the form `"hh:mm:ss"` with three components: hours, minutes, and seconds. The bounds on the time string are `00:00:00` and `23:59:59`.

However you choose to store the units of time is fine, as long as your class supports the remaining operations.

(b) Design the `int getNumberOfSeconds()` method, which returns the number of seconds that this `Time` object represents.

(c) Design the `int getNumberOfMinutes()` method, which returns the number of minutes that this `Time` object represents. If there are an inexact number of minutes, simply return the minutes. As an example, `new Time("02:30:45").getNumberOfMinutes()` returns `150` and not `151`.

(d) Override the `public String toString()` method to return a stringified version of the time where the hours, minutes, and seconds are separated by colons. Single-digit units of time must contain a leading zero.

(e) Override the `public boolean equals(Object o)` method that returns whether a given `Time` object represents the same time as `this` instance.

(f) Design the `void addTime(Time t)` method that adds a given `Time` object to `this` instance. Take the following invocations as an example.

```
Time t1 = new Time("02:30:45");
Time t2 = new Time("11:45:18");
Time t3 = new Time("00:53:57");
t1.add(t2);
t1.toString(); => 14:16:03
t1.add(t3);
t1.toString(); => 15:00:00
```

(g) Design the `void increment(String u)` method, which receives a string u denoting the unit of time to increment. If u is not one of `"HOUR"`, `"MINUTE"`, or `"SECOND"`, return `false`, and otherwise return `true`. Take the following invocations as examples. Remember to account for fringe cases, e.g., incrementing a unit that is about to roll-over to the next.

```
Time t1 = new Time("02:30:45");
t1.increment("HOUR");
t1.toString(); => "03:30:45"
t1.increment("MINUTE");
t1.toString(); => "03:31:45"
```

```
t1.increment("SECOND");
t1.toString(); => "03:31:46"
```

Exercise 1.21. (⋆⋆)
This exercise has six parts.

In this question you will implement the `MiniStack` data structure. This is similar to the `MiniArrayList` class from the chapter, but, of course, is a stack and not an array list.

Unlike many stack implementations, however, we will use an array-backed stack. This means that, instead of using a collection of `private` and `static` `Node` classes, the stack will use an array to store its elements. When the array runs out of space, a new one is allocated and the elements are copied over.

(a) First, design the generic `MiniStack` class. Its constructor should receive no arguments, and instantiate two instance variables: `T[] elements` and `size` to a new array and zero respectively. Remember that you cannot instantiate a generic array, so how do we do that? The initial capacity of the array should be set to `INITIAL_CAPACITY`, which is a `private static final` variable declared in the class as ten.
(b) Second, design the `void add(T t)` method, which adds an element onto the top of the stack. The "top of the stack," when using an array, is the right-most element, i.e., the element with the highest index. It might be a good idea to design a `private` helper method that resizes the underlying array when necessary. Your resize factor, i.e., how you resize the stack, is up to you.
(c) Third, design the `T peek()` method, which returns (but does not remove) the top-most element of the stack.
(d) Fourth, design the `T pop()` method, which returns *and* removes the top-most element. Be sure that your `add` method still works after designing `pop`.
(e) Fifth, design the `int size()` method, which returns the number of logical elements in the stack.
(f) Finally, override the `public String toString()` method to return a string containing the elements of the stack from top-to-bottom, separated by commas and a space. For example, if the stack contains, from bottom-to-top, `10`, `20`, `30`, `40`, and `50`, the `toString` method returns `"50, 40, 30, 20, 10"`.

Exercise 1.22. (⋆⋆⋆)
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 1.23. (⋆⋆⋆)
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 `private PQueue<T> copy()` method that returns a new queue with the same elements as the current queue. You should divide this method into a case analysis: one where `this` queue is empty and another where it is not. In the former case, return a new queue with no elements. In the latter case, iterate over the elements of the queue, enqueuing each element into a new queue. You will need instantiate a new `Node` (reference) for each element.
(d) 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. You should use the `copy` method to your advantage.
(e) Design the `PQueue<T> dequeue()` method that removes the first element of the queue, returning a new queue without this first value. You should use the `copy` method to your advantage.
(f) Design the `T peek()` method that returns the first element of the queue.
(g) Design the `static <T> 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.

(h) Design the int size() method that returns the number of elements in the queue. You should not traverse the queue to compute this value.

Exercise 1.24. (⋆⋆⋆)
This exercise has three parts.

A *deterministic finite state automaton* is an extremely primitive machine that represents transitions between the different states of a system. Think, as an example, of a light switch; there is an "OFF" state and an "ON" state, where flipping the switch flops between the two. The switch flip represents the input that causes a transition from one state to another. Programming languages most often use finite automata for character recognition, i.e., what characters are valid in the language grammar. The following is an example of a DFA diagram that accepts input strings that contain an odd number of 'a' characters from an input alphabet $\Sigma = \{a, b\}$.

(a) First, begin by designing the skeleton for the DFA class, which contains the following private and static class definitions:

- State, which stores a string identifier, an "isStart" boolean flag and an "isFinal" flag. The class should contain appropriate accessors but no mutators.
- Transition, which stores two State objects a and b representing the "from" and "to", as well as the required input to transition from a to b.

(b) The DFA constructor should be empty, and the class definition should store a Set<Transition> as well as a Set<State>.

(c) Design the void addState(State s) method, which adds a new State to the finite automaton.

(d) Design the State transition(State s, String i) method, which returns the state arrived after making the transition from s via input i.

(e) Finally, design the boolean accepts(String v) method, which receives an input string v and traverses over the automaton to determine if it accepts or rejects the input. We accept v if the last state we end on is marked as a final state.

Exercise 2.25. (⋆⋆⋆)
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:

(a) 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.

(b) boolean isReflexive() returns true if the relation is reflexive. A relation $\mathcal{R}$ is reflexive if, for all $x \in S$, $\langle x, x\rangle \in \mathcal{R}$.

(c) 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 $\langle y, x\rangle \in \mathcal{R}$.

(d) `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 $\langle x, z\rangle \in \mathcal{R}$.
(e) `boolean isEquivalence()` returns true if the relation is an equivalence relation. A relation $\mathcal{R}$ is an equivalence relation if it is reflexive, symmetric, and transitive.
(f) `boolean isIrreflexive()` returns true if the relation is irreflexive. A relation $\mathcal{R}$ is irreflexive if, for all $x \in S$, $\langle x, x\rangle \notin \mathcal{R}$.
(g) `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 $x \neq y$.
(h) `boolean isAsymmetric()` returns true if the relation is asymmetric. A relation is asymmetric if it is both antisymmetric and irreflexive.
(i) `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 $\langle x, y\rangle \in \mathcal{R}$.
(j) `Set<Pair<String, String>> reflexiveClosure()` returns a set representing the reflexive closure of a binary relation, which is $\mathcal{R} \cup r(\mathcal{R})$, where r returns a reflexive set over S.
(k) `Set<Pair<String, String>> isSymmetricClosure()` returns a set representing the symmetric closure of a binary relation, which is $\mathcal{R} \cup s(\mathcal{R})$, where r returns a symmetric set over S.
(l) `Set<Pair<String, String>> transitiveClosure()` returns a set representing the transitive closure of a binary relation, which is $\mathcal{R} \cup t(\mathcal{R})$, where t returns a transitive set over S.

As an added optimization, we should cache whether the current relation is one of these properties when prompted. If we do not add a pair to the relation, then it makes little sense to recompute whether or not is, say, reflexive. Implement this as an optimization, however you wish, into the class.

Exercise 1.26. ($\star\star$)
This exercise has five parts. Repeated string concatenation is a common performance issue in Java. As we know, Java `String` objects are immutable, which means that concatenation creates a new `String` objects. This is fine for small strings, but for larger strings (or concatenation operations performed in a loop), this can be a performance bottleneck. Each concatenation requires copying the entire string. Java provides the `StringBuilder` class to alleviate the issue. In this exercise, you will design a `MiniStringBuilder` class that mimics the behavior of `StringBuilder`. You cannot use `StringBuilder` or the older `StringBuilder` classes in your implementation.

(a) Design the `MiniStringBuilder` class, which stores a `char[]` as an instance variable. The class should also store a variable to keep track of the number of "logical characters" that are in-use by the buffer.
(b) Design two constructors for the `MiniStringBuilder` class: one that receives no arguments and initializes the default capacity of the underlying `char[]` array to 20, and another that receives a `String` s and initializes the `char[]` array to the characters of s.
(c) Design the `void append(String s)` method, which appends the given string s onto the end of the current string stored in the buffer. The given string should not simply be appended onto the end of the buffer, but rather added to the end of the previous string in the buffer. If the buffer runs out of space, reallocate the array to be twice its current size, similar to how we reallocate the array in the `MiniArrayList` example class.
(d) Design the `void clear()` method, which resets the `char[]` array to the default size of 20.
(e) Override the `public String toString()` method, which returns the `char[]` array as a `String` object.

Exercise 1.27. (⋆⋆)
This exercise has twelve parts. A complex number $c \in \mathbb{C}$ has two components: a real number a and an imaginary number b. Together, these components compose into $a + bi$. In this exercise you will design a class that operates over complex numbers.

(a) Design the `ComplexNumber` class, whose constructor receives two `double` values: a and b. Store these as instance variables.
(b) Design the empty constructor that initializes a and b to 0 and 0 respectively.
(c) Implement the respective accessor and mutator methods for the real and imaginary components.
(d) Override the `public String toString()` method to return a string representation of the complex number of the form `"a + bi"` or `"a - bi"` when b is either positive or negative.
(e) Override the `public boolean equals(Object o)` method to compare two complex numbers. Of course, this entails comparing the real component and the imaginary component.
(f) Override the `public int hashCode()` method to return a hashcode that encodes the `a` and `b` components respectively.
(g) Design the `double magnitude()` method, which returns the magnitude of `this` complex number. The magnitude of a complex number is the square root of the sum of its components.
(h) Design the `double argument()` method, which returns the argument, or angle, of `this` complex number in radians. The argument of a complex number is computed as $\tan^{-1} b/a$.
(i) Design the `ComplexNumber conjugate()` method, which returns the conjugate of `this` complex number. The conjugate of a complex number flips the parity of the imaginary component. That is, if we have a complex number $a + bi$, its conjugate is $a - bi$.
(j) Design the `ComplexNumber add(ComplexNumber c2)` method, which receives a `ComplexNumber` as an argument and returns a new `ComplexNumber` representing the sum of `this` complex number and the given number. The sum of two complex numbers is the sum of the real components and the imaginary components.
(k) Design the `ComplexNumber sub(ComplexNumber c2)` method, which receives a `ComplexNumber` as an argument and returns a new `ComplexNumber` representing the difference of `this` complex number and the given number. The difference of two complex numbers is the difference of the real components and the imaginary components.
(l) Design the `ComplexNumber mul(ComplexNumber c2)` method, which receives a `ComplexNumber` as an argument and returns a new `ComplexNumber` representing the product of `this` complex number and the given number. The product of two complex numbers is as follows:

$$(a + bi)(c + di) = (ac - bd) + (ad + bc)i$$

Hint: use `add` and `mul` to your advantage.

Exercise 1.28. (⋆)
This exercise has 2 parts.

(a) Design the `Accumulator` class, which stores an instance variable of type `Number`. The `Accumulator` constructor receives a value of type `T` and stores it as an instance variable.
(b) Design the `apply` method, which receives a `Number` and adds it to the instance variable, then returns the instance variable. If `apply` has only received integers as arguments, then the result should be interpreted as an integer and not a floating-point value. We're recreating a challenge invented by Paul Graham called the "Accumulator Factory."

As an example, consider the following sequence.

```
Accumulator acc1 = new Accumulator(1);
acc1.apply(5);
acc1.apply(7);
assertEquals(13, acc1.apply(0));
assertEquals(15.3, acc1.apply(2.3));
```

Exercise 1.29. (⋆)
This exercise has 3 parts.

The *Kotlin* programming language supports customized *ranges*. That is, we can define an interval using dot notation, e.g., `1..10`, then query a value over that interval. For instance, `x in 1..10` returns whether or not `x` is between `1` and `10`, inclusive. This comparison, however, extends beyond primitive datatypes; ranges may operate over classes. For example, we can create a range `"hi".."howdy"`, which defines the range of strings in between `"hi"` and `"howdy"`.

(a) Design the generic `Range` class. It should store, as instance variables, a minimum and a maximum value, both of which are of type `<T extends Comparable<T>>`, meaning `T` must be a comparable type.
(b) The `Range` constructor should receive these two values as parameters and assign them to the instance variables accordingly.
(c) Design the `boolean contains(T v)` method that returns whether or not v is between the interval that this range operates over.

Exercise 1.30. (⋆⋆)
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 1.31. (⋆⋆)
This exercise has three parts. In this exercise, you'll be developing a `Document` interface along with its implementing classes:

- `TextDocument`
- `SpreadsheetDocument`
- `PresentationDocument`

The `Document` interface is defined as follows:

```
interface Document {

  /**
   * Returns the number of pages in this document.
   */
  int numberOfPages();

  /**
   * Returns a string representing that the Document
   * is being printed.
   */
  default String print() {
```

```
    return "Printing the document!";
  }
}
```

Notice that we have a `default` method, which is one that an implementing class does **not** have to implement. It provides "default" functionality, should the "implementee" not want to implement the method (hence the name!).

(a) Implement the other three classes with the following specifications:

- A `TextDocument` consists of 100 pages. When it is printed, it should return a message `"Printing text document!"`.
- A `SpreadsheetDocument` has 50 pages. When it is printed, it should return a message `"Printing spreadsheet document!"`.
- A `PresentationDocument` contains 20 pages. It utilizes the default implementation of the `print` method.

(b) Design the `PrintingOffice` class, which includes the following static method: `static OptionalDouble avgPages(List<Document> lodocs)`. This method calculates and returns the average number of pages across the provided list of `Document` objects. Remember why we use `Optional`: if there are no `Document` objects in the list, we would be dividing by zero if we took the average!

(c) Inside the `PrintingOffice` class, modify it to include the `static void printDocuments(List<Document> documents)` method, responsible for invoking the `print` method on each object in the list of `Document` instances.

Exercise 1.32. (⋆⋆)
This exercise has three parts.

(a) Design the `INumberFormat` interface, which contains one method: `String format(int n)`.

(b) Design the `DollarFormat` method, which implements `INumberFormat`, and returns a string where the number is prepended with a dollar sign `"$"`.

(c) Design the `CommaFormat` method, which implements `INumberFormat`, and returns a string where the number contains commas where appropriate. For example, `format(4412)` should return `"4,412"`.

Exercise 1.33. (⋆⋆⋆)
In the chapter, we described the `PizzaOrder` class. This exercise introduces readers to the visitor design pattern, which we explore in greater detail in **??**.

(a) First, design the `GroupOrder` class, which keeps track of multiple pizzas in an order. Store a `Queue<PizzaOrder>` as an instance variable and instantiate it to a `PriorityQueue`. The `GroupObject` constructor should receive a `Comparator<PizzaOrder`. Pass this to the `PriorityQueue` instantiation.

(b) Design the `ITopping` interface, which represents a topping. For now, it contains no methods. Then, design four classes: `Pepperoni`, `Onion`, `Pineapple`, and `Anchovie`, all of which implement `ITopping`.

(c) Now, let's design a class that allows us to do multiple actions with toppings. Design the generic `IToppingVisitor` class, which has four methods: `T visit(Pepperoni p)`, `T visit(Onion o)`, `T visit(Pineapple p)`, and `T visit(Anchovie a)`.

(d) Design the `ToppingPriceVisitor` class, which implements the interface `IToppingVisitor`, whose type parameter is a `Double`. The idea is that the `ToppingPriceVisitor` class serves as a way of associating a property with toppings without having to modify/amend the class definitions. Override the four methods to return `3.50`, `2.50`, `5.75`, and `4.00` respectively.
(e) Modify the `PizzaOrder` class to have its map instance variable associate `ITopping` objects to `Integer`, rather than `String` to `Integer`.
(f) Amend the `ITopping` interface to now supply the `<T> T visit(IToppingVisitor<T> v)` method. Its subtypes should override `visit` by defining it as a call to `v.visit(this)`, where `v` is the visitor object parameter.
(g) Finally, design the `PizzaOrderPriceComparator` class, which implements `Comparator<PizzaOrder>`, and compares pizzas based on the price of its toppings. Pizzas with a higher cost are prioritized over pizzas with a lower cost.

Exercise 1.34. (⋆⋆⋆)
A *lazy list* is one that, in theory, produces infinite results! Consider the `ILazyList` interface below:

```
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:[33]

```
class FactorialLazyList implements ILazyList<Integer> {

  private int n;
  private int fact;

  FactorialLazyList() {
    this.n = 1;
    this.fact = 1;
  }

  @Override
  public Integer next() {
    this.fact *= this.n;
    this.n++;
    return this.fact;
  }
}
```

Testing it with ten calls to `next` yields predictable results.

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

class FactorialLazyListTester {

  @Test
  void testFactorialLazyList() {
```

[33] We will ignore the intricacies that come with Java's implementation of the `int` datatype. To make this truly infinite (up to the system's memory limit), we could use `BigInteger`.

```
    ILazyList<Integer> FS = new FactorialLazyList();
    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 $f(1) = 1$.

```
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 1.35. ($\star\star$)
Design the `LazyListTake` class. Its constructor 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.

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

class LazyListTakeTester {

 @Test
 void testLazyListTake() {
  LazyListTake llt1 = new LazyListTake(new FactorialLazyList(), 8);
  LazyListTake llt2 = new LazyListTake(new FibonacciLazyList(), 10);

  assertAll(
```

```
    () -> assertEquals("[1, 2, 6, 24, 120, 720, 5040, 40320]",
                       llt1.getList().toString()),
    () -> assertEquals("[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]",
                       llt2.getList().toString()));
 }
}
```

Exercise 1.36. (⋆⋆)
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 **??**). 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.

```
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 1.37. (⋆⋆)
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 1.38. (⋆⋆⋆)
In this exercise you will design a simple particle system manager. A *particle system* is a data structure that manages particles, or small effects, in a graphical engine. Think of a video game that has smoke, fire, water, explosion, or other kinds of effects. In general, these all use particle engines for managing hundreds of thousands of particle objects. Therefore, such an engine should be efficient.

(a) In the first part of this exercise, you will design the `Particle` class.

(i) A `Particle` contains a `double x` and `double y` representing its position, a `double width` and `double height` representing its dimensions, and a `double dx` and `double dy` representing its velocity. Finally, it contains a `double life` representing its life. The constructor should receive these as parameters and assign them to the instance variables.

(ii) Inside the `Particle` class, design the `update` method, which adds the particle's velocity to its position. It should also decrement the `life` instance variable by one. If `life` ever becomes zero or negative, the particle is no longer alive. If the particle *isn't* alive, do not update its position (nor decrement its life).

(iii) Design the `isAlive` method that returns whether or not the particle is alive.

(b) Now, you will design the `ParticleSystem` class for efficiently managing multiple particle instances. The idea behind this particle system is that we create a *memory pool*, and poll already-allocated particles from it when available. That is, when a particle dies, it moves to the "dead" sector, but that memory still exists. Then, when we want to create a new `Particle`, we first check to see if there are any dead particles that we can reuse. If so, we reuse that particle's allocated memory and simply reassign variables.

(i) In the `ParticleSystem` class, store the following instance variables and instantiate them as `LinkedList` instances in the constructor. The constructor should also receive a value `maxAlive`, which is assigned to a `final int MAX_ALIVE` instance variable.

- `List<Particle> alive`, which stores the alive particles in the system. All particles in this list should be non-`null`.
- `List<Particle> dead`, which stores the dead particles in the system. All particles in this list should be non-`null`.

(ii) Design the `boolean addParticle(double x, double y, double w, double h, double dx, double dy, double life)` method that adds a particle to the system with the given parameters. If there are no dead particles available, then simply allocate a `new Particle` onto the rear of the `alive` list. If there is a dead particle, use that allocated space instead and assign the parameters to the object using the respective setters. Then, move the particle out of the `dead` list and onto the rear of the `alive` list. If it is impossible to add a new particle (because there is no space for more alive particles), return `false`. Otherwise, return `true`.

(iii) Design the `void updateSystem()` method that traverses over the alive particles, and invokes their `update` methods. After invoking a particle's `update` method, check to see if it is alive or not. If it is not alive, move it out of the `alive` list and into the `dead` list.

(c) In the final part of this exercise, you will design two "types of" particles.

(i) Design the `SparkParticle` class, which inherits from `Particle`. "Spark particles" move in a straight line, but their velocity decreases over time due to air resistance until they stop moving.

- The `SparkParticle` constructor receives the same values as its superclass counterpart.
- Override the `update` method to decrease the vertical and horizontal velocities by 10% with each call to `update`. Do *not* call `super.update()`. Instead, update the position of the particle directly inside this class. Remember that those variables are private in the `Particle` class.
- Override the `isAlive` method to return `false` when its horizontal and vertical velocity values are both less than 0.01 away from zero. Otherwise, it should return `true`.

(ii) Design the `SmokeParticle` class, which inherits from `Particle`. "Smoke particles" move in a straight line, but their velocity decreases over time due to air resistance until they stop moving.
- The `SmokeParticle` constructor receives the same values as its superclass counterpart.
- Override the `update` method to increase the width and height dimensions by 2% with each call to `update`. Do *not* call `super.update()`. Instead, update the position of the particle directly inside this class (the behavior is the same as the `Particle` superclass). Remember that those variables are private in the `Particle` class. Finally, decrement the life by `0.2` rather than `1`.

Exercise 1.39. ($\star$)
Design the `static <T> Predicate<T> orEq(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 1.40. ($\star\star$)
Design the `static <T> List<T> predOrEq(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 1.41. ($\star$)
Design the `static <T> boolean andMap(List<T> l, 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 1.42. ($\star\star$)
Design the `static <T, U> U foldr(List<T> ls, BiFunction<T, U, U> f, U u)` method that receives a list of values ls, a function f, and an initial value u. The method should return the result of folding the list from the right with the given function and initial value. By "folding," we mean that we apply f to the last element of the list and the initial value, then apply f to the second-to-last element and the result of the previous application, and so forth. To think of this in terms of infix notation over some list, consider the list $[a, b, c, d]$. Folding it over the function $\circ$ and initial value u is $a \circ (b \circ (c \circ (d \circ u)))$. Do *not* use the `reduce` method, as that method solves the problem we want *you* to solve!

Exercise 1.43. ($\star$)
Design the `static <T, U> List<U> buildList(int n, Function<T, U> func)` method that receives an integer n and a function f and returns a list of n elements, where the i^{th} element is $f(i)$. For example, if we invoke `buildList(5, x -> x * x)`, we should receive the list $[1, 4, 9, 16, 25]$.

Exercise 1.44. ($\star$)
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 1.45. ($\star\star$)
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 1.46. ($\star\star$)
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 1.47. ($\star\star$)
This exercise involves the interpreter we wrote in the chapter. Our interpreter, so far, is very memory inefficient. The reason is not apparent at first glance, but consider every time that we create a `NumberNode` or, especially, a `BooleanNode`. There are only two possibilities for a `BooleanNode`: true and false, which are immutable by design. So, the interpreter should never waste time allocating an instance thereof when it can simply reference an existing one. Such an optimization follows the "factory" design pattern, which we will explore in Chapter **??**.

Privatize the `BooleanNode` constructor, then design the `static BooleanNode of(boolean b)` method. It receives a boolean value v and returns a reference to a pre-allocated static true or false node. Similarly, also privatize the `NumberNode` constructor and design the `static NumberNode of(double v)` method. This method returns a new `NumberNode` if the given value v is not an integer between $[0, 1000)$. Otherwise, precache those integer values and store them in a private lookup table.

Exercise 1.48. ($\star$)
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 1.49. ($\star\star$)
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 1.50. ($\star\star$)
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 1.51. (⋆⋆)
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 **??**, 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 1.52. (⋆⋆⋆)
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: `AstNode apply(List<AstNode> 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 `AstNode` instances, and the second type quantified to `AstNode`. 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.

```
@FunctionalInterface
interface IFunction<T, R> {

  R apply(T t);
}
```

```
import java.util.Map;
import java.util.HashMap;

class PrimNode extends AstNode {

  private static final Map<String, IFunction<List<AstNode>, AstNode>> OPS;

  static {
    OPS = new HashMap<>();
    OPS.put("+", this.primPlus);
  }

  @Override
  AstNode eval(Environment env) { /* TODO. */ }

  private AstNode primPlus(List<AstNode> args, Environment env) {
    /* Details omitted. */
  }
}
```

Exercise 1.53. (⋆⋆⋆)
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 evaluated abstract syntax trees, which should be stored in a list.
   (ii) 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.
   (iii) Evaluate the function abstract syntax tree to get its function definition as an abstract syntax tree.
   (iv) 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!

```
new ProgramNode(
  new DefNode("!",
    new FuncNode(
      List.of("n"),
      new IfNode(
        new PrimNode("eq?",
          new VarNode("n"),
          new NumNode(0)),
        new NumNode(1),
        new PrimNode("*",
          new VarNode("n"),
          new ApplyNode("!",
            new PrimNode("-",
            new VarNode("n"),
            new NumNode(1)))))))),
  new ApplyNode("!", new NumNode(5)))
```

Exercise 1.54. (⋆⋆)
This exercise involves the interpreter we wrote in the chapter, and relies on the addition of `FuncNode` and `ApplyNode`. Our current version of the interpreter uses *dynamic scoping*.

A dynamically-scoped interpreter is one that uses the value of the closest declaration of a variable. This seems like nonsense without an example, so consider the following code listing.

```
(define f
  (let ([x 10])
    (λ ()
      x)))

(let ([x 3])
  (f))
```

Under dynamic scoping, this program outputs `3`, because the binding of `x` takes on the value `3`. On the other hand, if we were using a *lexically-scoped* interpreter, the program would output `10`, which seems to make more sense due to the binding that exists immediately above the function declaration. The question is: how do we implement lexical scoping into our interpreter? The answer is via *closures*, which are data structures that couple a function definition with an environment. Then, when we apply a closure to an argument (if it exists), we restore the environment that was captured by the closure.

To this end, design the `ClosureNode` class, whose constructor receives a `FuncNode` and an `Environment`. The respective `eval` method returns `this`, but `FuncNode` changes slightly. Rather than returning `this`, we return a `ClosureNode`, which wraps the current `FuncNode` and the environment passed to `eval`. Finally, inside `ApplyNode`, evaluating the function definition should resolve to a closure. Evaluate the arguments to the closure inside the passed environment, but *extend* the captured environment, and bind the formals to the arguments in this extended environment. The body of the closure's function definition is then evaluated inside this new environment.

Making this alteration not only causes our programs to output the "common sense" result, but also means we can implement recursive functions using *only* a `LetNode`. See if you can figure out how to do this!

Exercise 1.55. (⋆⋆⋆)
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, override `toString` inside `ConsNode`, which amounts to printing each element, separated by spaces, inside of brackets, e.g., $[l_0, l_1, \ldots, l_{n-1}]$.

Exercise 1.56. (⋆⋆)
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 1.57. (⋆⋆⋆)
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 1.58. (⋆⋆)
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 1.59. (⋆⋆⋆)
In Chapter **??**, 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.[34] 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.

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

class FactorialTailRecursiveTester {

  @Test
  void testFactTailRecursiveTrampoline() {
    assertAll(
      () -> assertEquals(BigInteger.valueOf(120),
                         factTailCall(BigInteger.valueOf(5), BigInteger.ONE)),
      () -> assertDoesNotThrow(StackOverflowException.class,
                               () -> factTailCall(BigInteger.valueOf(50000),
                                                  BigInteger.ONE)),
    )
  }
}
```

[34] We omit the driver method to shorten the code, as the important part lies inside the recursive implementation.

```
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 ac - the accumulator.
   * @return a tail call that is either done or not done.
   */
  static ITailCall<BigInteger> factTailCall(BigInteger n, BigInteger ac) {
    if (n.equals(BigInteger.ZERO)) {
      return TailCallUtils.done(ac);
    } else {
      return TailCallUtils.call(() ->
               factTailCall(n.subtract(BigInteger.ONE), ac.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:

- `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.
- `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 1.60. (⋆⋆⋆)
Recall the unification exercise from Chapter **??**. 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 1.61. (⋆⋆⋆)
In this series of problems, you will design several methods that act on very large *natural numbers* resembling the `BigInteger` class. You ***cannot*** use any methods from `BigInteger`, or the `BigInteger` class itself.

(a) Design the `BigNat` class, which has a constructor that receives a string. The `BigNat` class stores a `List<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 `public String toString()` method to return a string representation of the `BigNat` object.
(c) Design the `BigNat clone()` method that returns a new `BigNat` instance that contains the same number.
(d) Override the `public boolean equals(Object obj)` method to compare two `BigNat` values for equality. Remember that you have to cast the given parameter to an instance of the `BigNat` class.
(e) Implement the `Comparable<BigNat>` interface, and override the method that it provides, namely `public int compareTo(BigNat b)`, 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 the 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? Multiplication is defined as repeated addition. Think about the performance implications of this approach.
(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? Division is defined, for all intents and purposes over the natural numbers, as repeated subtraction. Think about the performance implications of this approach.

Exercise 1.62. (⋆⋆⋆)
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 **??**). 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 the `boolean isTautology(Node t)`, `boolean isContingency(Node t)`, and the `boolean isContradiction(Node t)` methods, which return whether or not the given statement is a tautology, contingency, or contradiction, 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.

2 Exceptions and I/O

Abstract With the foundations of Java, data structures, and classes/objects covered, we now move into more advanced topics, such as exception handling and I/O. This chapter will discuss unchecked and checked exceptions, as well as how to handle them. We will also discuss several means of working with file I/O, including more advanced topics such as serialization. Finally, the chapter ends with an explanation of more modern Java I/O techniques.

2.1 Exceptions

Exceptions, at their core, 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.

2.1.1 Unchecked Exceptions

We handle exceptions at either compile time or runtime. The exceptions themselves are thrown at runtime, but some 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. We should also note that the `RuntimeException` class serves as the superclass of all unchecked exceptions.

Example 2.1. Consider what happens when a program contains code that may or may not divide a numeric value by zero. If the bad division operation occurs, Java automatically throws an `ArithmeticException` with a relevant explanation of the problem, that being a divide-by-zero. The exception halts program execution at the point thereof, but what's interesting is that we can control the behavior of an unchecked exception through a `try/catch` block. Within a `try` block, we include the code that potentially raises the exception. In the associated `catch` block, we declare a variable for the exception we aim to catch, such as `ArithmeticException`, and then manage it within that block. Let's write a method that does nothing more than divides the sum of two numbers by the third.

```
import java.lang.ArithmeticException;

class ArithmeticExceptionExample {

  double div(int a, int b, int c) {
    return (a + b) / c;
  }

  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 it makes more sense in this scenario to throw the exception and halt program execution, rather than propagating a zero up to the caller. Another solution might be to return an `Optional` from the method, but the `Optional` class is more about compositionality of stream methods rather than exceptions.

Example 2.2. In the preceding example, we catch the `ArithmeticException` that Java throws. Though, suppose we have a situation in which *we* want to throw the exception. Because the `div` problem arises from a bad parameter, we might wish to throw an `IllegalArgumentException`, which designates exactly what its name suggests. We insert a conditional check to test 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 raise the exception. Should we want to signal that as a hint, the method signature may 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 unit test a new version of `div` by determining whether 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 *what* exception the code might throw as a reference to the class definition.[1] Additionally, the argument must be passed as an executable argument. Remember though have worked with executable constructs before via lambda/anonymous functions! Simply wrap the code that might raise an exception inside a lambda, and everything works as expected.

[1] To reference a class definition as an object, we access `.class` on the class as if it were a static method.

```
import static Assertions.assertDoesNotThrow;
import static Assertions.assertThrows;

import java.lang.IllegalArgumentException;

class IllegalArgumentExceptionExampleTester {

  @Test
  void testIllegalArgumentException() {
    assertAll(
      () -> assertDoesNotThrow(div(5, 3, 1)));
      () -> assertThrows(IllegalArgumentException.class, () -> div(5, 3, 0)),
  }
}
```

```
import java.lang.IllegalArgumentException;

class IllegalArgumentExceptionExample {

  int 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 calling the `.getMessage` method on the exception object, which is helpful for producing custom error messages/responses or redirecting the message to a different destination.

```
import java.lang.IllegalArgumentException;

class IllegalArgumentExceptionExample {

  int 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 2.3. Arrays and strings both produce unchecked exceptions when incorrectly indexed via `ArrayIndexOutOfBoundsException` and `StringIndexOutOfBoundsException` respectively, both of which inherit from the `IndexOutOfBoundsException` class. We imagine that these have both been received, in great frustration, from the readers a indeterminate number of times. An index out of bounds exception stems from accessing data beyond the permissible bounds of some collection or structure.

```
import static Assertions.assertAll;
import static Assertions.assertDoesNotThrow;
import static Assertions.assertThrows;

import java.lang.StringIndexOutOfBoundsException;

class IndexOutOfBoundsExceptionExampleTester {

  @Test
  void testOobException() {
    String ex1 = "String";
    int[] ex2 = new int[]{5, 3, 1, 2, 4, 6};
    assertAll(
      () -> assertDoesNotThrow(() -> ex1.charAt(0)),
      () -> assertDoesNotThrow(() -> ex1.charAt(ex1.length() - 1)),
      () -> assertDoesNotThrow(() -> ex2[0]),
      () -> assertDoesNotThrow(() -> ex2[ex2.length - 1])
      () -> assertThrows(StringIndexOutOfBoundsException.class, () -> ex1.charAt(17)),
      () -> assertThrows(StringIndexOutOfBoundsException.class, () -> ex1.charAt(-1)),
      () -> assertThrows(ArrayIndexOutOfBoundsException.class, () -> ex2[17]),
      () -> assertThrows(ArrayIndexOutOfBoundsException.class, () -> ex2[-1]));
  }
}
```

Another uncomfortably common unchecked exception that many Java programmers encounter is the `NullPointerException`. The `NullPointerException` most often discovered when referencing an object that has yet to be instantiated, or was accidentally never instantiated at all.

Example 2.4. Casting an object of type τ_1 to an incompatible type τ_2 results in an unchecked `ClassCastException`. By "an incompatible type," we mean to say that the object is either not an instance of the τ_2 type, or τ_1 and tau_2 are not in a superclass/subclass relationship. Primitive datatypes are not subject to this exception, as they are not objects.[2] All primitive datatypes, minus booleans, can be casted 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` or vice-versa. We can treat `List<T>` as an `AbstractList<T>` by performing a cast, e.g., `AbstractList<T> x = (AbstractList<T>) ls;`, where `ls` is defined as being of type `List<T>`.

Example 2.5. Sometimes, a program can reach a state where continuing is impossible or illogical. In these circumstances, we can throw an `IllegalStateException`, designating that the program has reached a point that it should not under normal pretenses. An example is attempting to access a closed `Scanner` instance.

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

[2] No pun intended.

```
in.close();
String s = in.nextLine(); // Throws IllegalStateException!
```

2.1.2 Checked Exceptions

A *checked exception* is one that the programmer must explicitly handle. Java will fail to compile a program that does not enclose a checked exception type within a `try/catch` block, or when the method signature does not specify that it throws the exception. Almost all checked exceptions arise from I/O operations, such as reading from or to a data source, so further elaboration at this point the discussion at this point is not particularly helpful. We will discuss checked exceptions in the context of I/O operations in the following (non-sub)section.

2.1.3 User-Defined Exceptions

We can define our own exceptions in terms of other exceptions. Exceptions are nothing more than class definitions, which may be extended/inherited.

Example 2.6. Consider defining the `BadStringInputException` class, which inherits from `RuntimeException` . We might define `BadStringInputException` as an exception that Java throws when, after reading the user's input, we find that the input is not a "alpha string," i.e., a string that contains only letters. Let's define a constructor that takes a string as an parameter, serving as the exception message.[3]

```
class BadStringInputException extends RuntimeException {

  BadStringInputException(String msg) {
    super(String.format("BadStringInputException: %s", msg));
  }
}
```

Then, if we write code that reads a string from the user (through standard input), we can throw a `BadStringInputException` if said input is a non-alphabetic string. The following code segment uses the `matches` method, which receives a regular expression and returns whether the string satisfies the expression. More specifically, `[a-zA-Z]+` states that there must be at least one lowercase or uppercase character in the provided string. A method that calls `readAlphaString` does not need to handle the exception, as it unchecked and will be caught by the runtime system.

```
import java.util.Scanner;

class BadStringInputExceptionExample {

  static void readAlphaString() {
    Scanner in = new Scanner(System.in);
    String s = in.nextLine();
    if (!s.matches("[a-zA-Z]+")) {
      throw new BadStringInputException(s);
    }
```

[3] We note that, broadly speaking, creating new types of exceptions is rarely beneficial, because Java provides a plethora of exception definitions that cover most use cases.

```
  }
}
```

2.2 File I/O

Presumably this section discusses the syntax and semantics of file input and output. Although this is correct, we will explain reading data from "non-plain-text" sources such as websites and network connections through sockets.

2.2.1 Primitive I/O Classes

Example 2.7. Let's write a program that reads data from a file and echos it to standard output.

```
import java.io.IOException;
import java.io.FileNotFoundException;
import java.io.FileInputStream;

class FileInputStreamExample {

  public static void main(String[] args) {
    FileInputStream fis = null;
    String inFile = "file1.in";
    try {
      fis = new FileInputStream(inFile);
      // 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", inFile);
    } catch (IOException ex) {
      System.err.printf("main: I/O err: %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 line(s) that possibly throws the checked exception within a `try/catch` block. Namely, in the preceding code, the `FileInputStream` constructor is defined to potentially throw a `FileNotFoundException`, whereas its `read` method throws a generalized `IOException` if an input malfunction 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 invoke `.close` on the instance to release the allocated system resources and deem the file no longer available.[4] Expanding upon the `finally` block a bit more, 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, locks, that otherwise may not be released in the event of an exception or program redirection.[5] 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, Java provides the *try-with-resources* construct, which autocloses the resource.[6]

Example 2.8. Let's use the try-with-resources block to copy the contents of one file into another. In essence, we will design 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.

```
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); }
    } catch (FileNotFoundException ex) {
      System.err.printf("main: could not find file1.in\n");
    } catch (IOException ex) {
      System.err.printf("main: I/O err: %s\n", ex.getMessage());
    }
  }
}
```

The file input/output stream classes receive/send data as raw bytes from their source/destination streams. In most instances, we probably want to read *characters* from a data source or to a data destination rather than the raw bytes themselves. 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 textual data, 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, examples include `.pdf`, `.docx`, and so forth.

Example 2.9. Using `FileReader`, we will once again write an "echo" program to read data from its file source and writes its contents to standard output. Of course, we may want to output data to a file, in which case we use the dual to `FileReader`, that being `FileWriter`. In summary, `FileWriter` provides several methods for writing strings and characters to a data destination. In the following example we will also write some data to a test file, then examine its output based on our choice of method invocations.

```
import java.io.*;
```

[4] We can check whether an input stream is available via the `.available` method.

[5] A lock is a construct used in concurrent programs, which we will exemplify in **??**.

[6] 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.

```
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 `file2.out`, we see that it contains `"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 coerces (valid) integers 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 calling these low-level operations is expensive on the CPU because they read individual bytes, sequentially, which is horribly inefficient. The `BufferedReader` and `BufferedWriter` classes aim to alleviate this problem by utilizing data buffers. A better approach is to read data in chunks, rather than byte-by-byte, to reduce the number of operating system-level calls. When the allocated buffer is full, the data within is flushed to either the source or destination. By chunking the data in this manner, we reduce the number of times that the program interacts with the operating system, which consequently reduces the CPU overhead. To read from a stream using buffers, we use the `BufferedReader` class. Its constructor receives a `Reader` instance, which may be one of several classes. For example, to read from a file, we provide a `FileReader` to the `BufferedReader` constructor. 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 buffers, we use the analogous `BufferedWriter` class, which receives a `Writer` instance.

Example 2.10. Using `BufferedReader` and `BufferedWriter`, we will write a program that reads data from a file and outputs it to another file.

```
import java.io.*;

class BufferedReaderWriterExample {

  public static void main(String[] args) {
    try (BufferedReader br = new BufferedReader(
                             new FileReader("file1.in"));
```

BufferedReader Methods

The `BufferedReader` class provides methods for reading from a data source using a buffered mechanism.

`BufferedReader` *R* `= new BufferedReader(new FileReader(`*f*`))` creates a new buffered reader instance that reads from the file *f*, where *f* is either a `String` or a `File` object.
`int` *R*`.read()` reads a single character from the input stream *R*. Calling `read` advances the location of the file pointer by one byte. If the stream is empty or reads an `EOF` character, returns −1.
`String` *R*`.readLine()` reads a line of text from the input stream *R*. Calling `readLine` advances the location of the file pointer to the next line. If the stream is empty or has no further lines to consume, returns `null`.
`void` *R*`.close()` closes the input stream *R*.

Fig. 2.1: Useful `BufferedReader` Methods.

BufferedWriter Methods

The `BufferedWriter` class provides methods for writing to a data source using a buffered mechanism.

`BufferedWriter` *W* `= new BufferedWriter(new FileWriter(`*f*`))` creates a new buffered writer instance that writes to the file *f*, where *f* is either a `String` or a `File` object.
`void` *W*`.write(`*s*`)` writes a string *s* to the output stream *W*.
`void` *W*`.close()` closes the output stream *W*.

Fig. 2.2: Useful `BufferedWriter` Methods.

```
        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 execution time of buffered I/O to non-buffered I/O operations. The following code shows two implementations of reading the contents of a very large file and echoing them to another. We have defined two 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!

```
import java.io.*;

class PerformanceExamples {

  private static void buffered() {
    try (BufferedReader br = new BufferedReader(
```

PrintWriter Methods

The `PrintWriter` class provides utility methods for writing different types of data to a data destination.

`P = new PrintWriter(new FileWriter(f))` creates a new print writer instance that writes to a file f, where f is either a `String` or a `File` object.
`void P.print(x)` writes the string representation of x to the output stream P.
`void P.println(x)` writes the string representation of x to the output stream P, followed by a newline character.
`void P.printf(f, x)` writes a formatted string to the output stream P, where f is a format string and x is the value to be formatted.
`void P.close()` closes the output stream P.

Fig. 2.3: Useful `BufferedReader` and `BufferedWriter` Methods.

```
                            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 performance 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 2.11. Using `PrintWriter`, let's output some arbitrary constants and formatted strings to a file.

```
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 prone to mistakes and requires updating the "stringification" any time a field is added, removed, or altered. Instead, we can use the `ObjectOutputStream` and `ObjectInputStream` classes, which *serialize* and *deserialize* objects respectively. 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.

Example 2.12. 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. `Serializable` is a *marker interface*, meaning that it has no methods to implement. Instead, it is a "flag," of sorts, that informs the compiler that the class can be serialized. 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; the data within is intended to be read only by a program.

```
import java.io.Serializable;

class ObjectStreamExample {
  // ... previous code not shown.

  private static class Player implements Serializable {

    private String name;
    private int score;
    private int health;
    private double[] topScores;

    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,
```

Scanner Constructor Methods

The `Scanner` class has several constructors for reading from different data sources.

`S = new Scanner(System.in)` instantiates a scanner that reads from the standard input stream.
`S = new Scanner(f)` instantiates a scanner that reads from the file f, where f is a `File` object.
`void S.close()` closes the input scanner S.

Fig. 2.4: Useful `Scanner` Constructors.

```
                        topScores=%s]", name, score, health,
                        Arrays.toString(topScores));
    }
  }
}
```

Suppose, on the contrary, that we want to 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 pre-existing classes.

Example 2.13. In **??**, 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 representing `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.

```
class Employee {

  private long employeeId;
  private String firstName;
  private String lastName;
  private double salary;
  private boolean fullTime;

  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 returns a `List<Employee>` 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
```

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

import java.util.List;

class EmployeeScannerTester {

  @Test
  void testReadRecords() {
    List<Employee> emps = 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"));
  }
}
```

```
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.
   */
  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));
      }
```

Scanner Querying Methods

The `Scanner` class has several methods for determining the type of data that is next in the input stream.

`boolean` S`.hasNext()` returns `true` if the scanner has another token in its input.
`boolean` S`.hasNextInt()` returns `true` if the scanner has another integer in its input.
`boolean` S`.hasNextDouble()` returns `true` if the scanner has another double in its input.
`boolean` S`.hasNextBoolean()` returns `true` if the scanner has another boolean in its input.
`boolean` S`.hasNextLine()` returns `true` if the scanner has another line in its input.

Fig. 2.5: Useful `Scanner` Querying Methods.

Scanner Methods

The `Scanner` class has several methods for reading different types of data from its input stream.

`String` S`.next()` returns the next token from the scanner. Any leading whitespace is skipped. Generally, this method should not be used, instead opting for one of the four methods below.
`int` S`.nextInt()` returns the next integer from the scanner. If there is a newline character following the integer, it is left in the buffer. If there is no integer to be read, throws an `InputMismatchException`.
`double` S`.nextDouble()` returns the next double from the scanner. If there is a newline character following the double, it is left in the buffer. If there is no double to be read, throws an `InputMismatchException`.
`boolean` S`.nextBoolean()` returns the next boolean from the scanner. The same rules apply as for `nextInt` and `nextDouble`.
`String` S`.nextLine()` returns the next line from the scanner. The newline character is removed from the input buffer, but *not* included in the returned string.

Fig. 2.6: Useful `Scanner` Methods.

```
    } 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 2.14. Let's write a program that reads a file containing integers, then 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.

```
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);
    }

    String inFile = args[0];
    String outFile = inFile.substring(0, inFile.lastIndexOf('.')) + ".out";

    try (Scanner f = new Scanner(new File(inFile));
         PrintWriter pw = new PrintWriter(new FileWriter(outFile))) {
      while (f.hasNextInt()) {
        int val = f.nextInt();
        if (val % 2 == 0) {
          pw.println(val);
        }
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }
}
```

Example 2.15. 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]`. We can write JUnit tests to verify that our program works as intended.

```
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));
  }
}
```

```
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.
   */
  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 2.16. 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 the following:

```
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.

```
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;
    } else {
      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 outFile - name of output file.
   */
  private static void concatenate(String[] files, String outFile) {
    try (BufferedWriter bw = new BufferedWriter(
                             new FileWriter(outFile))) {
      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[] inFiles = Arrays.copyOfRange(args, 0, args.length - 1);
    String outFile = args[args.length - 1];

    if (!sameExtension(inFiles)) {
      System.err.println("ConcatenateFiles: bad extensions");
      System.exit(1);
    } else {
      concatenate(inFiles, outFile);
    }
  }
}
```

2.3 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 2.17. 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. Repeatedly writing these almost identical lines of code is 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 and store them in a `List<String>`.

```
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 2.18. Unfortunately, `readAllLines` is extremely memory-inefficient, requiring us to store a list of every line in the file. 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 (Morling, 2023).

To start the exercise, 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 either updates an existing `Measurement` in the map or inserts 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.

```
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.
```

```
   */
  static String computeMeasurement(String filename) {
    // TODO.
    return null;
  }

  private static class Measurement {

    private double min, max, total;
    private int noOccurrences;

    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.
     */
    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. 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, then 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, 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.

```
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.
   */
  static String computeMeasurement(String filename) {
    Map<String, Measurement> mMap = new HashMap<>();
    try (Stream<String> lines = Files.lines(Path.of(filename))) {
      lines.forEach(x -> {
        String[] arr = x.split(";");
        String location = arr[0];
        double temp = Double.parseDouble(arr[1]);
        Measurement ms = mMap.putIfAbsent(location,
                                          new Measurement(temp));
        if (ms != null) { ms.add(temp); }
      });
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
    return mMap.keySet()
               .stream()
               .sorted()
               .map(s -> String.format("%s=%s", s, mMap.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.

Example 2.19. Our last example of working with File I/O is a simple Sudoku solver. *Sudoku* is a game where the objective is to fill each row, column, and sub-grid with exactly one of possible entries, generally from 1 to 9. There are nine 3×3 subgrids that form a square, which results in a 9×9 grid.

The most straightforward way to mechanically solve a Sudoku puzzle is via a backtracking algorithm. That is, we try to place a number in a cell and, if it leads to success, we continue with the solution. Otherwise, we undo the placement and try again. We will use I/O to read in a partial Sudoku puzzle and to write the solution out to another file.

Let's write the `SudokuSolver` class, whose constructor receives a file that represents a partial Sudoku puzzle. The input specification contains nine rows and nine columns, with dots to denote a missing number. From here, we will design the `boolean solve()` method, which returns whether or not a solution exists. If there is one, it is stored in an instance variable of the class. We will also design the `void output(String fileName)` method to output the solution to a file. If there is no possible solution, the program will throw an `IllegalStateException` to indicate a failure.

```
import java.io.IOException;
import java.nio.file.*;
import java.util.*;

class SudokuSolver {

  private static final int N = 9;
  private int[][] board;
```

```
  private int[][] solution;

  SudokuSolver(String filename) {
    this.board = new int[N][N];
    this.solution = new int[N][N];
    try (Stream<String> lines = Files.lines(Path.of(filename))) {
      int row = 0;
      lines.forEach(x -> {
        for (int i = 0; i < x.length(); i++) {
          this.board[row][i] = x.charAt(i) == '.' ? 0 : x.charAt(i) - '0';
          this.solution[row][i] = this.board[row][i];
        }
        row++;
      });
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }

  boolean solve() {
    /* TODO. */
    return false;
  }

  void output(String filename) {
    /* TODO. */
  }
}
```

Our `solve` method jump-starts a backtracking algorithm that attempts to solve the puzzle using recursion. Let's design a private helper method to receive the row r and column c of the cell to fill. If r and c are both equal to N, then we have reached the end of the board and therefore have a solution. Otherwise, we need to find the next empty cell to fill. This is a three-step process:

(i) First, if the y coordinate is equal to N, then we have reached the end of the row and need to move onto the next.
(ii) If the cell is not empty, we move onto the next cell.
(iii) If the cell is empty, we try to place a number in the cell. If the number is valid, we continue with the solution. Otherwise, we undo the placement and try again.

What does it mean for a number to be valid? A number is valid in its placement if it does not already exist in the row, column, or subgrid. Let's write another private helper method that, when given a cell at row r and column c, and a number n, determines whether or not the number is valid.

```
import java.io.IOException;
import java.nio.file.*;
import java.util.*;

class SudokuSolver {
  // ... previous code not shown.

  SudokuSolver(String filename) { /* Implementation omitted. */ }

  /**
   * Returns whether or not a number is valid in a given cell.
   * @param r - row of cell.
```

```
   * @param c - column of cell.
   * @param n - number to place in cell.
   * @return true if number is valid, false otherwise.
   */
  private boolean isValid(int r, int c, int n) {
    // Check the row and column simultaneously.
    for (int i = 0; i < N; i++) {
      if (this.board[r][i] == n || this.board[i][c] == n) {
        return false;
      }
    }

    // Check the subgrid.
    int sr = (r / 3) * 3;
    int sc = (c / 3) * 3;
    for (int i = sr; i < sr + 3; i++) {
      for (int j = sc; j < sc + 3; j++) {
        if (this.board[i][j] == n) {
          return false;
        }
      }
    }
    return true;
  }
}
```

From this we can begin to work on the recursive backtracking algorithm, using the outline we provided earlier.

```
import java.io.IOException;
import java.nio.file.*;
import java.util.*;

class SudokuSolver {
  // ... previous code not shown.

  SudokuSolver(String filename) { /* Implementation omitted. */ }

  /**
   * Returns whether or not a solution exists. If a solution does not
   * exist, the variable that stores the solution is set to null.
   * @return true if a solution exists, false otherwise.
   */
  private boolean solve() {
    if (solve(0, 0, this.solution)) {
      return true;
    } else {
      this.solution = null;
      return false;
    }
  }

  /**
   * Recursive backtracking algorithm to solve the puzzle.
   * @param r - row of cell.
   * @param c - column of cell.
   * @param sol - solution array.
   * @return true if we have a solution, and false if the current
   * placement is invalid or leads to a "dead end".
   */
```

```
  private boolean solve(int r, int c, int[][] sol) {
    if (r == N) { return true; }
    else if (c == N) { return solve(r + 1, 0, sol); }
    else if (this.board[r][c] != 0) { return solve(r, c + 1, sol); }
    else {
      for (int i = 1; i <= N; i++) {
        if (isValid(r, c, i)) {
          this.sol[r][c] = i;
          if (solve(r, c + 1, sol)) { return true; }
          this.sol[r][c] = 0;
        }
      }
    }
    return false;
  }
}
```

Finally, the `output` method is straightforward. We use a `PrintWriter` to write the solution to a file. If there is no solution (meaning the solution instance variable is set to `null`), then we throw an `IllegalStateException`.

```
import java.io.IOException;
import java.nio.file.*;
import java.util.*;

class SudokuSolver {
  // ... previous code not shown.

  SudokuSolver(String filename) { /* Implementation omitted. */ }

  /**
   * Outputs the solution to a file. The solution is just a 9x9 grid of
   * numbers, and does not attempt to format the output in any way.
   * @param filename - name of output file.
   */
  void output(String filename) {
    try (PrintWriter pw = new PrintWriter(new FileWriter(filename))) {
      if (this.solution == null) {
        throw new IllegalStateException("No solution exists.");
      } else {
        for (int i = 0; i < N; i++) {
          for (int j = 0; j < N; j++) {
            pw.print(this.solution[i][j]);
          }
          pw.println();
        }
      }
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
  }
}
```

2.4 Exercises

Exercise 2.1. (⋆⋆)
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 2.2. (⋆⋆)
Design the `Capitalize` class, which contains one `static` method: `void capitalize(String in)`. The `capitalize` method reads a file of sentences (that are not necessarily line-separated), and outputs the capitalized versions of the sentences to a file of the same name, just with the `.out` extension (you must remove whatever extension existed previously).

You may assume that a sentence is a string that is terminated by a period and only a period, which is followed by a single space. 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 we invoke `capitalize("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 (*again, remember to remove the newline before "hopefully".*):

```
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 2.3. (⋆⋆)
Design the `SpellChecker` class, which contains one `static` method: `void spellCheck(String dict, String in)`. The `spellCheck` method reads two files: a "dictionary" and a content file. The content file 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 (line-separated) words. If a word is not spelled correctly, wrap it inside brackets `[]`.

Output the modified sentences to a file of the same name, just with the `.out` extension (you must remove whatever extension existed previously). You may assume that words are space-separated and that no punctuation exist. Hint: use a `Set`! 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 we invoke `spellChecker("dictio` `"file3a.in")`, 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 [throughlyy] [misssing]
[puncutiation]
```

Exercise 2.4. (⋆⋆)
Design the `OrderWebUrls` class, which contains one `static` method: `void orderWebUrls(String in)`. The `orderWebUrls` method reads in a file of line-separated web URLs. A web URL contains a protocol separated by a colon and two forward slashes, and a host name. For example, in the URL `https://www.joshuacrotts.us`, the protocol is `https` and the host name is `www.joshuacrotts.us`. The method should read in web URLs in this specific format and sort them, lexicographically, based on the hostname. If two hostnames are identical and only differ by the protocol, then the order becomes based on the protocol.

Exercise 2.5. (⋆⋆)
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 interface, 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 2.6. (⋆⋆)
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.

Exercise 2.7. (⋆⋆)
A common use for file input and output is data analysis. Design a class StatisticsDescriptor that has the following methods:

(a) 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.
(b) double mean(), which returns the mean of the dataset.
(c) double stddev(), which returns the standard deviation of the dataset.
(d) 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.
(e) double median(), which returns the median, or the middle value, of the dataset.
(f) double mode(), which returns the mode, or the most-frequent value, of the dataset.
(g) double range(), which returns the range, or the difference between the maximum and minimum values of the dataset.
(h) 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 **??**.
(i) 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 2.8. (⋆⋆)
You are teaching an introductory programming course and you want to keep a seating chart for your students. A seating chart is an arrangement of numbers $1..n$, the location in the classroom of which is defined by the instructor. Numbers that are lower in the range are closer to the front of the room. Design the SeatingChart class, which has the following methods:

(a) SeatingChart() is the constructor, which initializes the seating chart to be empty. The seating chart is represented as a List<Student>, where Student is a private and static class, inside SeatingChart, that you design. Students should have a name, a seat number,

and an accommodation parameter. The seat number is an integer, and the accommodation parameter is a boolean.

(b) `void read(String fileName)` reads in a list of students from a file into the seating chart. The file contains a list of students, one per line, with their name. A student also has an optional accommodation parameter, which means they should sit in a seat closer to the front of the room. The file is comma-separated, and if the student has an accommodation, it is represented by `true` after the student's name.

```
Alice
Bob,true
Charlie
```

(c) `void scramble()` scrambles the seating chart. That is, it randomly shuffles the students in the seating chart. This also accounts for the accommodations, so that students with accommodations are closer to the front of the room.

(d) `void alphabetize()` sorts the seating chart alphabetically by the students' names. This "mode" does not account for accommodations, and is thus strictly alphabetical.

(e) `List<Student> getStudents()` returns the seating chart as a list of students.

(f) `List<Student> getAccommodationStudents()` returns the students with accommodations as a list.

(g) `void output(String fileName)` outputs the seating chart to a file specified by the parameter. The file should contain the students' names and their seat numbers, one per line, separated by a comma. The output list should be in the order of the seating chart.

Exercise 2.9. (⋆⋆⋆)
You are designing a system for looking up car information, similar to that of, say, Kelly Blue Book or Carvana.

(a) First, design the `Car` class, which stores the make, model, color, trim, and VIN (vehicle identification number) of the car as strings, the year and number of prior owners as an integer, an enumeration that contains its title status (e.g., `Clean`, `Salvaged`, or `Rebuilt`), and its MSRP (in USD) as a floating-point value.[7]

(b) Make the `Car` class serializable, like we did in the chapter. That is, implement the `Serializable` interface and override the `writeObject` and `readObject` methods respectively.

(c) Override the `public boolean equals(Object o)` and `public int hashCode()` methods. Two `Car` objects are the same if they share the same VIN. When overriding `hashCode`, return a hash code that hashes all instance variables.

(d) Finally, override the `public String toString()` method, which returns a string similar to the following (with a tab character before each line):

```
    Make: Honda
    Model: Ac-
cord
    Color: Sil-
ver
    Trim: LX
    Year: 2007
    VIN: 1G4HDSLVRLX
```

[7] As a tip: if you are ever writing real-world software that works with currency values, you should *never* store currency as floating-point numbers, e.g., `double` or `float`. This is because of the inaccuracies that come with such representations in a computer. The preferred solution is to use an object type that separately stores cents and dollars such as `BigDecimal`.

```
Number of Previous Owners: 2
Title Status: Salvaged
MSRP: $20,475.00
```

(e) Now, design the `CarDatabase` class, whose (no parameter) constructor instantiates a `List<Car>` instance variable to store the list of cars. Then, design the following methods:

(i) `void addCar(Car car)`, which adds a car with the given values to the database.
(ii) `boolean removeCar(String vin)`, which removes a car with the given VIN. If the car was in the database, the method returns `true`, and `false` otherwise.
(iii) `boolean contains(String vin)`, which returns `true` if a `Car` with the given VIN exists in the database, and `false` otherwise.
(iv) `boolean contains(Car car)`, which returns `true` if the given car exists in the database, and `false` otherwise. Note that this method should be one line long and call the other variant of `contains`.
(v) `void readFile(String in)`, which populates the database with the `Car` objects from the given file. The file should contain only serialized `Car` objects and not plain-text.
(vi) `void writeFile(String out)`, which writes all cars in the database out to a file with the given file name. The file should contain only serialized `Car` objects and not plain-text.
(vii) `void sort(Comparator<Car> cmp)`, which sorts the database of `Car` objects according to the provided `Comparator` implementation.
(viii) `void sort()`, which sorts the database of `Car` objects according to their VIN.

Exercise 2.10. (⋆⋆)
You're interested in determining the letter statistics of a file containing text. In particular, you want to design a program that reports the frequency of each alphabetic character. Design the `LetterFrequency` class, which has the following methods:

(a) `LetterFrequency(String fileName)` is the constructor, which reads a file containing text into a `long[]` instance variable with 26 elements. Convert all upper-case letters into lower-case. Index 0 of the (frequency) array corresponds to `'a'`, and index 25 corresponds to `'z'`. Before reading the contents of the file, initialize the array to contain all zeroes.
(b) `void add(char c)` adds a character c to the frequency map. If c is not alphabetic, throw an `IllegalArgumentException`.
(c) `void add(String s)` calls the other `add` method on each character in the given string s.
(d) `long get(char c)` returns the frequency of a given character, which should be converted to lowercase. If c is not a letter, throw an `IllegalArgumentException`.
(e) `char get(int i)` returns the i^{th} most frequent character. If $i \notin [0, 25)$, throw an `IllegalArgumentException`.
(f) `List<Character> getMostFrequentChars(int n)` returns the n most frequent characters. If $n \notin [0, 25)$, throw an `IllegalArgumentException`. Hint: invoke the `get` method n times for values 1 to n inclusive.

Exercise 2.11. (⋆⋆)
This exercise has two parts. A *stack-based* programming language is one that uses a stack to keep track of variables, values, and the results of expressions. These types of languages have existed for several decades, and in this exercise you will implement such a language.

Design the `StackLanguage` class, whose constructor receives no parameters. The class contains two instance methods: `void readFile(String f)` and `double interpret()`.

- The `readFile` method reads a series of "stack commands" from the file. These can be stored however you feel necessary in the class, but you should not interpret anything in this method, nor should you throw any exceptions. You may want to create a private static class for storing commands.
- The `interpret` method interprets the stored list of instructions. If no instructions have been received by a `readFile` command, throw an `IllegalStateException`. Here are the following possible instructions:

  (a) `DECL` v X declares that v is a variable with value X.
  (b) `PUSH` X pushes a number X to the stack.
  (c) `POP` v pops the top-most number off the stack and stores it in a variable v. If v has not been declared, an `IllegalArgumentException` is thrown.
  (d) `PEEK` v stores the value at the top of the stack in the variable v. If v has not been declared, an `IllegalArgumentException` is thrown.
  (e) `ADD` X adds X to the top-most number on the stack.
  (f) `SUB` X subtracts X from the top-most number on the stack.
  (g) `XCHG` v swaps the value on the top of the stack with the value stored in variable v. If v has not been declared as a variable, an `IllegalArgumentException` is thrown.
  (h) `DUP` duplicates the value at the top of the stack.

  If the command is none of these, then throw an `UnsupportedOperationException`. You may assume that all commands, otherwise, are well-formed (i.e., they contain the correct number of arguments and the types thereof are correct). After interpreting all instructions, the value that is returned is the top-most value on the stack. If there is no such value, throw a `NoSuchElementException`.
  Hint: use a `Map` to store variable identifiers to values.

Exercise 2.12. (⋆⋆⋆)
Java provides many forms of input and output stream classes, e.g., `BufferedReader`/`BufferedWriter`. Unfortunately, it does not have classes, say `BitInputStream` and `BitOutputStream`, for outputting raw bits to a file. In this exercise you will implement these classes. [8]

(a) Design the `BitOutputStream` class, which extends `OutputStream`. It should store two instance variables: an instance of `OutputStream` and an array of eight integers. Each integer index corresponds to a bit to send to the output.

  (i) Design three `BitOutputStream` constructors: one that sets the stored output stream instance to `null`, a second that receives an `OutputStream` object and assigns it to the instance variable, and a third that receives a file name, and instantiates the stored output stream as a `FileOutputStream`. All three constructors should instantiate the array of "bits."
  (ii) Override the `public void flush() throws IOException` method from `OutputStream` to output the bits, as a single byte, to the file. This does *not* mean that you should output the raw `'1'` and `'0'` characters that are stored in the buffer. Instead, convert those bits into a single `int`, and write that value to the output stream. Hint: the bitwise operations `<<` and `|` may come in handy.
  (iii) Override the `public void write(int b) throws IOException` method from `OutputStream` to assign bit b to the next-free index in the array. If you run out of bits to store in the array, call `this.flush()`.

[8] This exercise is common in Java textbooks, and in our opinion, is worth repeating.

(iv) Design the `void writeBit(int b)` method, which adds a bit to the i^{th} index of the array. If the input b is not a 0 or 1, throw an `IllegalArgumentException`, otherwise call `this.write` with b.

(b) Design the `BitInputStream` class, which extends `InputStream`. This class should also store an instance of an `InputStream` as a field, as well as an array of bit values.

(i) Design three `BitInputStream` constructors that mimic the behavior of the `BitOutputStream` class constructors.
(ii) Design the `private int readBit()` method, which reads a bit from the buffer. Your code should call `read` on the input stream once every eight bits, i.e., every byte.
(iii) Override the `public int read()` method, which returns the next bit from the buffer. If you run out of bits to return, read a byte from the input stream. If there are no more bytes to read, return -1. Hint: `read()` itself returns -1 when there are no bytes remaining.

Exercise 2.13. ($\star\star\star$)
A maze is a grid of cells, each of which is either open or blocked. We can move from one free cell to another if they are adjacent. Design the `MazeSolver` class, which has the following methods:

(a) `MazeSolver(String fileName)` is the constructor, which reads a description of a maze from a file. The file contains a grid of characters, where `'.'` represents an open cell and `'#'` represents a blocked cell. The file is formatted such that each line is the same length. Read the data into a `char[][]` instance variable. You may assume that the maze dimensions are on the first line of the file, separated by a space.
(b) `char[][] solve()` returns a `char[][]` that represents the solution to the maze. The solution should be the same as the input maze, but with the path from the start to the end marked with `'*'` characters. The start is the top-left cell, and the end is the bottom-right cell. If there is no solution, return `null`.
We can use a backtracking algorithm to solve this problem: start at a cell and mark it as visited. Then, recursively try to move to each of its neighbors, marking the path with a `'*'` character. If you reach the maze exit, then return `true`. Otherwise, backtrack and try another path. By "backtrack," we mean that you should remove the `'*'` character from the path. If you have tried all possible paths from a cell and none of them lead to the exit, then return `false`. We provide a skeleton of the class below.
(c) `void output(String fileName, char[][] soln)` outputs the given solution to the maze to a file specified by the parameter. Refer to the above description for the format of the output file and the input `char[][]` solution.

```
class MazeSolver {

  private final char[][] MAZE;

  MazeSolver(String fileName) { /* TODO read maze from file. */ }

  /**
   * Recursively solves the maze, returning a solution if it exists,
   * and null otherwise. We use a simple backtracking algorithm
   * in the helper.
   * @return a solution to the maze, or null if it does not exist.
   */
  char[][] solve() {
```

```
    char[][] soln = new char[MAZE.length][MAZE[0].length];
    return this.solveHelper(0, 0, soln) ? soln : null;
  }

  /**
   * Recursively solves the maze, returning true if we ever reach
   * the exit. We try all possible paths from the current cell, if
   * they are reachable. If a path ends up being a dead end, we
   * backtrack and try another path.
   * @param r - the row of the current cell.
   * @param c - the column of the current cell.
   * @param sol - the current solution to the maze.
   * @return true if we are at the exit, false otherwise.
   */
  private boolean solveHelper(int r, int c, char[][] sol) {
    /* TODO. */
  }
}
```

Exercise 2.14. (⋆⋆⋆)
The `cut` program is a command-line tool for extracting pieces of text from data. For this exercise, you will implement a very basic version of the program that reads data from the terminal.

(a) First, add support for the `-c` $X, Y, \ldots, Z$ flag, which outputs the characters at positions $X, Y, \ldots, Z$ in each line. If any number is less than 1, throw an `IllegalArgumentException`.
(b) Second, add support for the `-c` X-Y flag, which outputs the characters between and including positions X and Y. This option should also work with comma separators.
(c) Third, add support for the `-c` X- and `-c` -Y flags, which print the characters from X to the end of the line, and all characters up to Y.
(d) Fourth, Add support for the `-d`D and `-f`X flags. The former serves as a single character delimiter, and the latter indicates that X is either an interval or a range of fields to print. The fields are delimited by D. Note that these two flags cannot be used without the other. The format of X mirrors that of the input to the $-c$ flag. If D does not exist on a line, then the entire line is printed.

Exercise 2.15. (⋆⋆⋆)
The `sort` program is a command-line tool for sorting input from a data source. For this exercise, you will implement a very basic version of the program that reads data from the terminal.

(a) First, allow the sort command to receive either a file or a list of data. If the `-d`D flag is passed, use D as the delimiter. The default for a file is a newline, and the default for a non-file is the space.
(b) Second, add support for the `-r` flag for reversed sorting.
(c) Third, add support for the `-i` flag for case-insensitive sorting.
(d) Fourth, add support for the `-c` flag for checking to see if a file is sorted. Reports the first occurrence of out-of-order sort.
(e) Fifth, add support the the `-n` flag for sorting the values as if they are numbers. Notice the difference between sorting `9, 10, 8` with and without this flag.
(f) Sixth, add support for the `-u` flag for removing duplicate values.

(g) Seventh, add support for the `-o` flag for outputting to the file specified immediately after.

Any of these flags should be composable with another, with the exception of `-o` whose output file is the next argument, and `-c`, which outputs any disorders to standard out.

Exercise 2.16. (⋆⋆⋆)
The `awk` program is a command-line tool for text parsing and processing. For this exercise, you will implement a very basic version of the program that reads data from the terminal. Be aware that this exercise is more in line with a mini-project.

(a) Add support for the `-F` flag that, when immediately followed by a delimiter, uses that delimiter as a "field separator" when parsing input lines. For example, `-F,` uses a comma as the delimiter.
(b) Add the `-h` flag that ignores the first row in all subsequent commands. This is particularly useful when working with files that have headers, e.g., comma-separated value files.
(c) Next, add the `'print ...'` command. That is, the user should be able to type an open brace, followed by `print`, then some data, then a closing brace, all enclosed by single quotes. The `print` command receives multiple possible values, including 'column labels', i.e., N, where N is a column number. For example, `awk -F, 'print $1' input.csv` should print the first column of each row in the input file.
(d) After getting the previous command to work, add support for inlined prefix operations in the `print` command. That is, suppose we want to print the sum of the second and fourth column of each row. To do this, we might write `awk -h -F, 'print (+ $2 $4)' input.csv`. For simplicity, you may assume that there are only four operations: `'+'`, `'-'`, `'*'`, and `'/'`.
(e) After getting the previous command to work, add multiple-argument support for `print`. That is, if we want to compute the product of the first three columns, output a string saying `" multiplied is "` , followed by the product, we could write `awk -h -F, 'print $1,$2,$3 " multiplied is " (* $1 $2 $3) '`. Note that the delimiter between the column labels must match that passed by `-F`, otherwise there is no separator.
(f) After getting the previous command to work, add support for conditional expressions. That is, suppose we want to print the second column of a row only if it has a value greater than 200. We can achieve this via `"awk -h -F, 'print $2(> $2 200)'`. If you *really* want a challenge, you can add support for inlining other arithmatic expressions into a conditional. For example, if we want to print the third column only if the sum of the first two columns exceeds 1000, we might write `awk -h -F, 'print $3(> (+ $1 $2) 1000)'`.
(g) Finally, add the `-v=N:V` flag that acts as a variable map that can be used in a print command. That is, suppose we want to create a variable called `val` and assign to it `30`. We can do this via `-v=val:30`, then reference it in a `print` via `$`, e.g., `awk -F, 'print $val'`.

Exercise 2.17. (⋆⋆⋆)
A thesaurus is, in effect, a dataset of words/phrases and information about those words/phrases. For example, a thesaurus may contain a word's definition, synonyms, antonyms, part-of-speech, and more. There are hundreds of collections online that researchers use for sentiment analysis, natural language processing, and more. In this exercise you will create a mini-thesaurus parser that allows the user to lookup information about a word/phrase.

(a) First, design the skeleton for the `Thesaurus` class. It should store a `Set<Word>` S as an instance variable.

(b) Design the private and static `Word` class inside the `Thesaurus` class body. A `Word` stores a `String` s and a `Map<String, List<String>>` M as instance variables. The string is the word itself, and the map is an association of information "categories" to a list of content. For example, we can create a `Word` that represents `"happy"`, with an association of `"synonym"` to `List.of("content", "cheery", "jolly")`. Design the respective getters and setters for these two instance variables.

(c) In the `Word` class, design the `boolean updateCategory(String c, String v)` method, which receives a category c and a value v, and updates the list mapped by c to now include v. If `c` did not previously exist for that `Word`, add it to the map and return `false`. On the other hand, if c did previously exist for that `Word`, update its association and return `true`.

(d) In the `Word` class, override the `equals` and `hashCode` methods to compare two `Word` objects for equality and generate the hash code respectively. Two `Word` objects are equal if they represent the same word.

(e) In the `Thesaurus` class, design the `List<String> getInfo(String c)` and `List<String> getInfo(String w, String c, int n)` methods, where the former calls the latter with `Integer.MAX_VALUE` as `n`. The latter, on the other hand, looks up w in S, and

- If `w` $\notin S$, return `null`.
- Otherwise, return n items from the category c of w. If n is `Integer.MAX_VALUE`, return the entire list.

The list returned by both methods must be immutable and not a pointer to the reference in the map. Notice that we overload `getInfo` to perform different actions based on the number (and type) of received parameters.

References

[1] 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.

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

[3] McCarthy, J. (1962). *LISP 1.5 Programmer's Manual*. The MIT Press.

[4] Morling, G. (2023).

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

[6] Stroustrup, B. (2013). *The C++ Programming Language*. Addison-Wesley Professional, 4th edition.

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

Index

`finally`, 139

access modifier, 2
accessor method, 3

checked exception, 137, 138
class, 1
classes, 1
collision resolution, 49
constructor, 2

default method, 85
design pattern, 13
downcast, 55

encapsulation, 3
Euclid's algorithm, 25

final class, 76

hash code, 10
hash table, 49

impure, 30
in-place sorting, 69
initialize, 3
instance variables, 1
instantiation, 4

marker interface, 143
method overloading, 6
mutator method, 30

object, 1
object composition, 13
object initialization, 3
object instantiation, 4

persistent data structure, 114
polymorphic, 77
polymorphism, 77

serialization, 143
setter method, 30
shadow, 3
side-effect, 30
singleton, 13
static variable, 8

try-with-resources, 139

unchecked exception, 133

variable shadowing, 3