Joshua Crotts

Learning Java

A Test-Driven Approach

May 13, 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 bulletproof 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 Testing and Java Basics 1
- 1.1 A First Glimpse at Java 1
  - 1.1.1 The Main Method 8
- 1.2 Strings 8
- 1.3 Standard I/O 13
- 1.4 Randomness 16
- 1.5 Exercises 19

References 23

Index 24

Chapter 1
Testing and Java Basics

Abstract This chapter introduces readers to the basics of Java programming and testing. We will cover the fundamentals of Java syntax, including variables and data types. At the same time, we discuss the importance of test-driven development and how to write test cases in Java for methods. Our emphasis on testing makes work of the fact that testing is a critical part of writing correct and robust software. Test-driven development, in particular, is utilized as a "design recipe," of sorts, to guide programmers to solve a task.

1.1 A First Glimpse at Java

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

Example 1.1. Our program will consist of a function that converts a given temperature in Fahrenheit to Celsius.

```java
class TempConverter {

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

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

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

TempConverter, giving rise to believe that the class does something related to temperature conversion.

We wrote the fToC method *signature*, whose *return type* is a double, and has one *parameter*, which is also a double. A double is a floating-point value, meaning it potentially has decimals. For our method, this choice makes sense, because if we were to instead receive an integer int, we would not be able to convert temperatures such as 35.5 degrees Fahrenheit to Celsius.

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

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

Inside its *body* lies a single return, in which we return 0.0. The *return value* is what a *method call*, or *method invocation*, resolves to. For example, if we were to call fToC with any arbitrary double value, the call would be substituted with 0.0. Passing a value, or an *argument*, to a method is sometimes called *method application*.

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

The fToC method is meaningless without a complete implementation. Before we write the rest of the method body, however, we should design *test cases* to ensure it works as expected. Test cases verify the correctness (or incorrectness) of a method. We, as the readers, know how to convert a temperature from Fahrenheit to Celsius, but telling a computer to do such a conversion is not as obvious at first glance. To test our methods, we will use the *JUnit* testing framework. Creating tests for fToC is straightforward; we will make a second class called TempConverterTester to house a single method: testFToC.

```
class TempConverterTester {

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

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

When writing tests, it is important to consider *edge cases* and all possible branches of a method implementation. Edge cases are inputs that are possibly missed by an implementation, e.g., -40, since it is the same in both Fahrenheit and Celsius, or 0. So, let us add a few more test cases.[2]

[2] To condense our code, we exclude the TempConverter class name out of conciseness. On the other hand, we can omit the Assertions class name by importing the two methods as shown in the listing.

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

class TempConverterTester {

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

Computers are, unfortunately, not perfect; floating-point operations may result in precision/rounding errors. So, as an optional third argument to `assertEquals`, we can provide a *delta*, which allows for precision up to a certain amount to be accepted as a valid answer. For example, our tolerance for the fourth and fifth test cases is `0.01`, meaning that if our actual value is less than $\pm$ 0.01 away from the expected value, the test case succeeds.

Now that we have copious amounts of tests, we can write our method definition. Of course, it is trivial and follows the well-known mathematical definition.

```java
class TempConverter {

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

The definition of `fToC` brings up a few points about Java's type system. The primitive mathematics operations account for the types of its arguments. So, for instance, subtracting two integers will produce another integer. More noteworthy, perhaps, a division of two integers produces another integer, even if that result seems to be incorrect. Thus, `5 / 9` results in the integer `0`. If, however, we treat at least one of the operands as a floating-point value, we receive a correct result of approximately `0.555555`: `5.0 / 9`. Java by default uses the standard order of operations when evaluating mathematics expressions, so we force certain operations to occur first via parentheses.[3]

Unlike some programming languages that are *dynamically-typed*, e.g., Scheme, Python, JavaScript, the Java programming language requires the programmer to specify the types of variables (Sebesta, 2012, Chapter 5).[4] Java has several default *primitive datatypes*, which are the simplest reducible form of a variable. Such types include `int`, `char`, `double`, `boolean`, and others. Integers, or `int`, are any positive or negative number without decimals. Doubles,

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

[4] In Java 10, the `var` keyword was introduced, which automatically infers the type of a given expression (Gosling et al., 2023, Chapter 14).

or double, are values with decimals.[5] Characters, or char, are a single character enclosed by single quotes, e.g., 'X'. Finally, booleans, or boolean, are either true or false. There are other Java data types that specify varying levels of precision for given values. Integers are 32-bit signed values, meaning they have a range of $[-2^{31}, 2^{31})$. The short data type, on the contrary, is 16-bit signed. Beyond this is the byte data type that, as its name suggests, stores 8-bit signed integers. Floating-point values are more tricky, but while double uses 64 bits of precision, the float data type uses 32 bits of precision.

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

```java
class VectorDistance {

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

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

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

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

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

class VectorDistanceTester {

  @Test
  void testComputeDistance() {
    assertAll(
      assertEquals(8.66, computeDistance(3, 2, 1, 8, 7, 6), .01),
      assertEquals(12.20, computeDistance(0, 0, 0, 8, 7, 6), .01),
      assertEquals(8.30, computeDistance(-8, -2, 1, 0, 0, 0), .01));
  }
}
```

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

[5] The double data type cannot represent all real numbers. For example, the value π is an irrational (real) number, and thus cannot be represented exactly in a finite number of bits.

```java
class VectorDistance {

  /**
   * Computes the distance between two given vectors:
   * v_1=(x_1, y_1, z_1) and v_2=(x_2, y_2, z_2).
   * @param x1 - x component of first vector.
   * @param y1 - y component of first vector.
   * @param z1 - z component of first vector.
   * @param x2 - x component of second vector.
   * @param y2 - y component of second vector.
   * @param z2 - z component of second vector.
   * @return distance between v1 and v2.
   */
  static double computeDistance(double x1, double y1, double z1,
                                double x2, double y2, double z2) {
    return Math.sqrt(Math.pow(x1 - x2, 2)
                   + Math.pow(y1 - y2, 2)
                   + Math.pow(z1 - z2, 2));
  }
}
```

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

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

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

```java
class SlopeIntercept {

  /**
   * Computes the slope of the line represented by the two points.
   * @param x1 - x-coordinate of point 1.
   * @param y1 - y-coordinate of point 1.
   * @param x2 - x-coordinate of point 2.
   * @param y2 - y-coordinate of point 2.
   * @return slope of points.
   */
  static double slope(int x1, int y1, int x2, int y2) { return 0.0; }

  /**
   * Computes the y-intercept of the line represented by the two points.
   * @param x1 - x-coordinate of point 1.
   * @param y1 - y-coordinate of point 1.
   * @param x2 - x-coordinate of point 2.
   * @param y2 - y-coordinate of point 2.
   * @return y-intercept of line represented by points.
   */
  static double yIntercept(x1, int y1, int x2, int y2) { return 0.0; }
}
```

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

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

class SlopeInterceptTester {

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

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

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

```java
class SlopeIntercept {

  /**
   * Computes the slope of the line represented by the two points.
   * @param x1 - x-coordinate of point 1.
   * @param y1 - y-coordinate of point 1.
   * @param x2 - x-coordinate of point 2.
   * @param y2 - y-coordinate of point 2.
   * @return slope of points.
   */
  static double slope(int x1, int y1, int x2, int y2) {
    return (y2 - y1) / (x2 - x1);
  }

  /**
   * Computes the y-intercept of the line represented by the two points.
   * @param x1 - x-coordinate of point 1.
   * @param y1 - y-coordinate of point 1.
   * @param x2 - x-coordinate of point 2.
   * @param y2 - y-coordinate of point 2.
   * @return y-intercept of points.
   */
  static double yIntercept(x1, int y1, int x2, int y2) {
    return y1 - slope(x1, y1, x2, y2) * x1;
  }
}
```

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

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

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

```java
class QuarticFormulaSolver {

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

Test cases are certainly warranted, but may be a bit tedious to compute by hand, so we recommend using a verified calculator to compute expected solutions![6]

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

class QuarticFormulaSolverTester {

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

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

```java
class QuarticFormulaSolver {

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

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

And, as expected, all tests pass. With only methods and math operations at our disposal, the capabilities of said methods are quite limited.

1.1.1 The Main Method

Java programming tutorials are quick to throw a lot of information at the reader/viewer, and our textbook is no exception to this practice. Unfortunately, Java is a verbose programming languages, and to begin designing a method, we must wrap its definition inside a class. After this step, we can design the `static` method implementation. Readers no doubt question the significance of `static`. For the first few chapters, we will intentionally omit its definition, as an explanation would almost certainly confuse the reader coming from another language. Therefore, for the time being, simply view it as six characters, plus a space, that must be typed in order to write a method that we can then test with JUnit.

Imagine, however, a situation where we want to output the result of an expression without using a test, as we will do in many examples. Java requires a `main` method in any executable Java class that does not use tests. For the sake of completeness, let's now write the traditional "Hello, world!" program in Java using the `main` method instead of tests.

```
class MainMethod {

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

Yikes, that is a lot of required code to output some text to the console; what does `public` mean, and what are those `[]` brackets after the `String` word? Once again, we will not detail their importance, but view them as more mandatory characters to type when writing a `main` method. The only word we *will* explain is `void`, which means that the method does not return a value. If the readers are coming from a functional programming language, e.g., Scheme/Racket or OCaml, then it is almost certainly the case that they never worked with methods that did not return a value nor received no arguments.[7] The `println` method, for example, has no return value; its significance comes from the fact that it outputs text to the terminal/console, which is a *method side-effect*. We'll come back to what all of this means in subsequent chapters, but we could not avoid at least briefly mentioning it and its existence in the Java language.

1.2 Strings

Strings, as you might recall from another programming language, are sequences of characters. Characters are enclosed by single quotes, e.g., `'x'`. A Java `String` is enclosed by double quotes. e.g., `"Hello!"`. Strings may contain any number of characters and any kind of character, including no characters at all or only a single character. There is an apparent distinction between `'x'` and `"x"`: the former is a `char` and the latter is a `String`. Note the capitalization on the word `String`; the case sensitivity is important, because `String` is not one of the primitive datatypes that we described in the previous section. Rather, it is a sequence of `char`

[7] A method of no arguments is called a *thunk*, or a *nullary* method.

String Class

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

S_1 `+` S_2 adds the characters from S_2 onto the end of S_1, producing a new string.
`int` S`.length()` returns the number of characters in S.
`char` S`.charAt(`i`)` retrieves the $(i+1)^{\text{th}}$ character in S. We can also say that this retrieves the character at index i of S.
`String` S`.substring(`i`,` j`)` returns a new string containing the characters from index i, inclusive, to index j, exclusive. The number of extracted characters is $j-i$. We will use the notation $S' \sqsubseteq S$ to denote that S' is a substring of S.
`String` S`.substring(`i`)` returns a new string from index i to the end of S.
`int` S`.indexOf(`S'`)` returns the index of the first instance of S' in S, or -1 if $S' \not\sqsubseteq S$.
`boolean` S`.contains(`S'`)` returns `true` if $S' \sqsubseteq S$; `false` otherwise.
`String` S`.repeat(`n`)` returns a new string containing n copies of S.
`String String.valueOf(`v`)` returns a stringified version of v, where v is some primitive value.
`int Integer.parseInt(`S`)` returns the integer representation of a string S if it can be parsed as such.

Fig. 1.1: Useful String Methods.

values coalesed into one value "under-the-hood," so to speak. We can declare a `String` as a variable using the keyword combined with a variable name, just as we do for primitives.

```java
class NewStringTests {

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

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

Figure 1.1 states that strings are immutable, so how can we possibly combine them without altering one or the other? When concatenating two strings s_1 and s_2, Java creates a new string that is the result of the concatenation, leaving s_1 and s_2 unaltered.[8]

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

Comparing strings for equality seems straightforward: we can use == to compare one string versus another. Using == for determining string equality is a common Java beginner pitfall! Strings are *objects* and cannot be compared for value-equality using the == operator. Introducing this term "value-equality" insinuates that strings can, in fact, be compared using ==, which is theoretically correct. The problem is that the result of a comparison using == com-

[8] Hence, a series of repeated string concatenations is inefficient. The performance implications of this are addressed in a subsequent chapter.

pares the memory addresses of the strings. In other words, s_1 == s_2 returns whether the two strings reference, or point to, the same string in memory.

```java
class NewStringTests {

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

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

```java
class NewStringTests {

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

If we want, for some reason, to circumvent Java's string caching capabilities, we need a way of *instantiating* a new string for our variables to reference. We use the power of `new String` to create a brand new, non-cached string reference. We treat this as a method, of sorts, called the *object constructor*. In Chapter **??**, we will reintroduce constructors in our discussion on objects and classes, but for now, consider it (namely `new String`) as a method for creating distinct `String` instances. This method is *overloaded* to receive either zero or one parameter. The latter implementation receives a `String`, whose characters are copied into the new `String` instance. If we pass a string literal to the constructor, it copies the characters *from* the literal into the new string. At this point, the only possible object to be equal to s_1 is s_1 itself or another string that points to its value.

```java
class NewStringTests {

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

Example 1.5. One final point to make about string equality is that string concatenation is not *always* subject to the same optimization as string literals. Consider the following code snippet, which creates two string literals s_1 and s_2. Then, we concatenate s_1 and s_2 into a string s_3, and separately concatenate the string literals themselves into the string s_4. What we observe is that s_3 and s_4 do not point to the same reference, even though they contain the same characters.

```java
class NewStringTests {

  public static void main(String[] args) {
    String s1 = "Hello";
    String s2 = "World";
    String s3 = s1 + s2;
    String s4 = "Hello" + "World";
    System.out.println(s3 == s4); // false
  }
}
```

On the other hand, consider the following code, where we create two strings s_1 and s_2 to both reference the concatenation of the string literals `"x"` and `"y"`. Comparing s_1 and s_2 for object equality confusingly produces `true`. At compile-time, Java optimizes (and evaluates) the string literal concatenations, which leads to it being pooled to reference the same (string) object in memory.

```java
class NewStringTests {

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

Example 1.6. What if we want to compare strings based on their character content, rather than by memory reference? The `String` class provides a handy `equals` method that we invoke on instances of strings. Two strings s_1 and s_2 are equal if they are *lexicographically equal.* In essence, this is a long and scary word to represent the concept of "containing the same characters." Lexicographical comparisons are case-sensitive, meaning that uppercase and lowercase letters are report an unequal comparison.

```java
class StringLexicographicallyEqual {

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

Let's veer into the discussion on the lexicographical ordering of strings. Like numbers, strings are comparable and can be, e.g., "less than" another. According to Java (as well as most other programming languages), one string is less than another if it is lexicographically less than another, and this idea extends to all (string) comparison operations; not just equality. Memorizing the string ordering rules is cumbersome, so we propose the S.N.U.L. acronym. In general, special characters (S) are less than numbers (N), which are less than uppercase characters (U), which are less than lowercase characters (L).[9] For a full description with the modifications to our generalization, view an ASCII table, which provides the numerical equivalents of all standard American keyboard characters.

Example 1.7. Java returns the distance between the first non-equal characters in a string using `compareTo`. For instance, `"hello".compareTo("hi")` returns `-4` because the distance from the first non-equal characters, those being `'e'` and `'i'`, is minus four characters, since `'i'` is greater than `'e'`. If we compare `"hello"` against `"Hello"`, we get 32, because according to the ASCII table, `'h'` corresponds to the integer 104, and `'H'` corresponds to the integer 72; their difference is $104 - 72$. Comparing `"hello"` against `"hello"` produces zero, indicating that they contain the same characters.

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

Example 1.8. To retrieve the character at a given index, we invoke `charAt` on the string: `"hello".charAt(1)` returns `'e'`. Attempting to index out of bounds with either a negative number or a number that is equal to or exceeds the length of the string results in a `StringIndexOutOfBoundsException` exception.[10]

Example 1.9. We can use `indexOf` to find the index of the first occurrence of some value in a string. To demonstrate, `"hello, how are you?".indexOf("are")` returns `11` because the substring `"are"` occurs starting at index 11 of the provided string. If the supplied argument is not present in the string, `indexOf` returns -1.

Example 1.10. A *substring* of a string s is a sequence of subsequent characters inside s. We can extract a substring s' from some string s using the `substring` method: `"abcde".substring(2, 4)` returns `"cd"`. Note that the last index is exclusive, meaning that, as a rule of thumb, the number of characters returned in the substring is equal to the second argument minus the first argument. There is also a handy second version of this method that receives only one argument: it returns the substring from the given index up to the end of the string. For example, `"abcdefg".substring(1)` returns everything but the first character, namely `"bcdefg"`.

Example 1.11. We can convert between datatypes, such as an integer to a string and vice versa, using the `String.valueOf` and `Integer.parseInt` methods respectively. Passing any non-string primitive value as an argument to `valueOf` converts it into its string counterpart. Oppositely, if we pass a string that represents an integer to `parseInt`, we obtain the corresponding integer. The `Double.parseDouble` and `Boolean.parseBoolean` methods behave

[9] We say "in general" because certain characters that we might view as special, e.g., `'@'`, have a larger numeric value than numbers, e.g., `'2'`.

[10] An *exception* in Java is a type of "error" that occurs while a program is running. We put "error" in quotes because "errors," at least in Java, are unpredictable and are usually a sign of a severe problem in the program such as running out of system memory.

similarly. Passing an invalid string, i.e., one that does not represent the respective datatype, results in Java throwing a `NumberFormatException` exception.

```
String s1 = String.valueOf(1234);    // "1234"
String s2 = String.valueOf(Math.PI); // "3.141592653589793"
int n1 = Integer.parseInt(s1);       // 1234
double n2 = Double.parseDouble(s2);  // 3.141593
```

1.3 Standard I/O

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

First, we need to discuss *standard data streams*. Java (and the operating system in general) utilizes three standard data streams: standard input, standard output, and standard error. We can think of these as sources for reading data from and writing data to. The *standard output stream* is often accessed using the `System.out` class and through its various methods, e.g., `println`, `print`, and `printf`. To output a line of data to standard output, we invoke `System.out.println` with a string (or some other datatype that is coerced into a string). For relaying messages to the user in a terminal-based application, or even when debugging, outputting information to standard output is a good idea. On the other hand, sometimes a program fails or the programmer wants to output an error message. It is possible to output error messages to standard output, since they are otherwise indistinguishable. Though, Java has a dedicated *standard error stream* for outputting error messages and logs via `System.err`. Now, let's discuss `printf` in more detail due to its inherent power.

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

Example 1.12. To output an `int` or `long` using `printf`, we use the `%d` format specifier.

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

Example 1.13. To output a `double` using `printf`, we use the `%f` format specifier. We can also specify the number of digits `n` to print after the decimal point by using the format specifier `.%nf`. Note that floating a decimal to *n* digits does not change the value of that variable; rather, it only changes its string/output representation.

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

There are many ways to get creative with `printf`, including space padding, number formatting, left/right-alignment, and more. We will not discuss these in detail, but instead we provide Table 1.2 of the most common format specifiers.

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

Fig. 1.2: Common Format Specifiers

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

Example 1.14. Suppose we want to read an integer from the standard input stream. To do so, we first need to instantiate a `Scanner`, which declares a "pipe," so to speak, from which information is read. It is important to state that, while a `Scanner` may read from the standard input stream, it can also read from other input streams, e.g., files or network connections. We will explore this further in subsequent chapters, but for now, let's declare a `Scanner` object to read from standard input.

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

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

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

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

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

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

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

Example 1.15. Let's reimplement Python's `input` function, which receives a `String` serving as a prompt for the user to enter data. To make it a bit more user-friendly and elegant, we will add a colon and a space after the given prompt. Because we open a `Scanner` to read from the standard input stream, there is no need to worry about, say, calling `nextInt` prior to invoking `input`. If, on the other hand, we declared a static global `Scanner` that reads standard input, and we use that to read an integer *inside* `input`, we would be in trouble.

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

Example 1.16. Suppose we want to design a method that reads three Cartesian points, as integers, from standard input, and computes the area of the triangle that comprises these points. We can type all integers on the same line, as separated by spaces, because `nextInt` only parses the *next* integer in the input stream, delimited by whitespace. And, as we said before, `nextInt` skips over existing trailing spaces in the input stream buffer, so those spaces are omitted. From there, we use the formula for computing the area of the triangle from those points.

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

```
import java.util.Scanner;

class ThreePointArea {

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

We make a note that reading data from a scanner inside a static method to compute some value is not a very good idea. A better alternative solution is to read the data inside a different, unrelated method, e.g., `main`, then call `computeThreePointArea` with the six arguments representing each point.

```java
import java.util.Scanner;

class ThreePointArea {

  /**
   * Computes the area of a triangle given three
   * Cartesian points as parameters.
   * @param x1 - x coordinate of first point.
   * @param y1 - y coordinate of first point.
   * @param x2 - x coordinate of second point.
   * @param y2 - y coordinate of second point.
   * @param x3 - x coordinate of third point.
   * @param y3 - y coordinate of third point.
   * @return the area of the triangle.
   */
  static double computeThreePointArea(double x1, double y1,
                                      double x2, double y2,
                                      double x3, double y3) {
    return (x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1 - y2)) / 2.0;
  }

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

1.4 Randomness

So-called “true” randomness is difficult to implement from a computability standpoint. Thus, for most intents and purposes (i.e., all of those described in this textbook), it is sufficient to use *pseudorandomess* when generating random values. A *pseudorandom number generator* (PRNG) computes appearingly random values using a deterministic algorithm, which means that the output values from the generator are predictable. Although it might be incredibly difficult to predict pseudorandomly-generated numbers, it is theoretically possible, which makes those numbers insufficient and insecure for cryptographic schemata and algorithms. For designing, perhaps, a word-guessing game that randomly chooses words from a list, it is perfectly reasonable to use a pseudorandom number generator.

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

Example 1.17. Using `Random`, let’s generate an integer between 0 and 9, inclusive on both bounds. To do so, we first need to instantiate a `Random` object, which we will call `random`.

Then, we should invoke `nextInt` on the `random` object with an argument of 10. Passing the argument n to `nextInt` returns an integer $x \in [0, n-1]$.[11]

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

Example 1.18. Imagine we want to generate an integer between -50 and 50, inclusive on both bounds. The idea is to generate an integer between 0 and 100, inclusive, then subtract 50 from the result.

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

Example 1.19. When creating a `Random` object, we can pass a *seed* to the constructor, which is an integer that determines the sequence of pseudorandom numbers generated by our `Random` instance. Therefore, if we pass the same seed to two distinct `Random` objects, they will generate the same sequence of pseudorandom numbers. If we do not pass a seed to the constructor, then the `Random` object will use the current system time as the seed. In theory, we could select a predetermined seed to write JUnit tests for methods that rely on randomness.

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

import java.util.Random;

class DualRandomTester {

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

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

Example 1.20. Java's `Math` class provides a `random` method, which receives no arguments. To do anything significant, we must understand how this method works and what kinds of values it can return. The `Math.random()` method returns a random `double` between $[0, 1)$, where the upper-bound is exclusive. So, we could see numbers such as `0.391283114421`, `0`, `0.999999999999`, but never exactly one. We can use basic multiplicative offsets to convert this range into one that we might want. For example, to generate a random double value between $[0, 10)$, we multiply the output by ten, e.g., `Math.random() * 10`.

Example 1.21. To generate a random integer between -5 and 15, inclusive, using the `Math.random` method, we need to create an offset similar to what we did in the `Random`

[11] Readers should be aware that this version of `nextInt` is different from the one provided by the `Scanner` class and cannot be used interchangably.

example. First, we multiply the result of `Math.random()` by 21 to generate a floating-point value between $[0, 21)$. *Casting* (i.e., explicitly treating the returned expression as another type) this expression to an integer produces an integer between $[0, 20]$. Finally, subtracting five therefrom gets us the desired range.

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

1.5 Exercises

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

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

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

Exercise 1.4. (⋆)
Design the `double gigameterToLightsecond(double gm)` method, which converts a distance in gigameters to light seconds (i.e., the distance that light travels in one second). There are $1,000,000,000$ meters in a gigameter, and light travels $299,792,458$ meters per second.

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

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

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

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

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

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

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

Exercise 1.10. ($\star$)
Design the `double pyramidSurfaceArea(double l, double w, double h)` method, which computes the surface area of a pyramid with a given base length l, base width w, and height h. The formula is

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

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

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

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

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

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

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

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

Exercise 1.13. ($\star$)
Design the `double logBase(double n, double b)` that, when given a number n and a base b, returns $\log_b(n)$. You will need to make use of the change-of-base formula, which we provide below (n is the number to compute the logarithm of, b is the old base, and b' is the new base).

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

Exercise 1.14. ($\star$)
Design the `double pdf(double mean, double stddev, double x)` that calculates the probability density of a given value x in a normal distribution with mean μ and standard devia-

tion σ. The formula for the probability density function (PDF) of a normal distribution is as follows:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-1/2(x-\mu/\sigma)^2}$$

Exercise 1.15. ($\star$)
Design the `double lawOfCosines(double a, double b, double th)` method that, when given two side lengths of a triangle a, b and the angle between those two sides θ in degrees, returns the length of the third side c. The formula, which is the law of cosines, is listed below. Hint: `Math.cos` receives a value in radians; we can convert a value from degrees to radians using `Math.toRadians`.

$$c = \sqrt{a^2 + b^2 - 2ab\cos\theta}$$

Exercise 1.16. ($\star$)
Design the `double angle(double a, double b, double c)` method that, when given three side lengths of a triangle a, b, and c, returns the angle θ opposite to that of c in degrees. The formula for this computation is listed below. Hint: arccosine in Java is `Math.acos`.

$$\theta = \cos^{-1}\left(\frac{a^2 + b^2 - c^2}{2ab}\right)$$

Exercise 1.17. ($\star$)
A physics formula for computing object distance displacement is

$$d = v_i t + 1/2at^2$$

where d is the final distance traveled in meters, v_i is the initial velocity, t is the time in seconds, and a is the acceleration in meters per second. Design the `double distanceTraveled(double vi, double a, double t)` method that, when given these variables as parameters, returns the distance that the object in question traveled.

Exercise 1.18. ($\star$)
Design the `String weekday(int d)` method that, when given an integer d from 1 to 7, returns the corresponding day of the week, with 1 corresponding to `"Monday"` and `"Sunday"` corresponding to 7. You **cannot** use any conditionals or data structures to solve this problem. Hint: declare a string containing each day of the week, with spaces to pad the days, and use `indexOf` and `substring`.

Exercise 1.19. ($\star$)
Design the `String flStrip(String s)` method that, when given a string, returns a new string with the first and last characters stripped. You may assume that the input string contains at least two characters.

Exercise 1.20. ($\star$)
Design the `double stats(double x, double y)` method that receives two `double` parameters, and returns a `String` containing the following information: the sum, product, difference, the average, the maximum, and the minimum. The string should be formatted as follows,

where each category is separated by a newline `'\n'` character. Assume that `XX` is a placeholder for the calculated result.

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

Exercise 1.21. (⋆)
Design the `String userId(String f, String l, int y)` method that computes a user ID based on three given values: a first name, a last name, and a birth year. A user ID is calculated by taking the the first five letters of their last name, the first letter of their first name, and the last two digits of their birth year, and combining the result. Your method should, therefore, receive two `String` parameters and an `int`. Do not convert the year to a `String`. Below are some test cases.

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

Exercise 1.22. (⋆)
Design the `String cutUsername(String email)` method that receives an email address of the form `X@Y.Z` and returns the username. The username of an email address is `X`.

Exercise 1.23. (⋆)
Design the `String cutDomain(String url)` method that returns the domain name of a website URL of the form `www.X.Z`, where `X` is the domain name and `Z` is the top-level domain.

Exercise 1.24. (⋆)
Design the `int nextClosest(int m, int n)` method that, when given two positive (non-zero) integers m and n, finds the closest positive integer z to m such that z is a multiple of n and $z \leq m$. For example, if $m = 67$ and $n = 15$, then $z = 60$.

Chapter 2
References

[1] Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A., Smith, D., and Bierman, G. (2023). *The Java Language Specification, Java SE 22 Edition.* Oracle, 1st edition.
[2] Sebesta, R. W. (2012). *Concepts of Programming Languages.* Pearson, 10th edition.

Index

`Math.random()`, 16
`Random`, 16
`Scanner`, 14
`nextLine`, 14
`printf`, 13

argument, 2

body, 2

cast, 18
casting, 18
class, 1
concatenate, 9
concatenation, 9
constructor, 10

delta argument, 3

edge case, 2
exception, 12

format characters, 13
format specifiers, 13
format string, 13
function, 1

instantiation, 10

JUnit, 2

line buffered, 14

method, 1
method application, 2
method body, 2
method call, 2
method invocation, 2
method overloading, 10
method signature, 2

parameter, 2
primitive datatypes, 3
pseudorandom, 16
pseudorandomness, 16

return type, 2
return value, 2

side-effect, 8
signature, 2
standard data streams, 13
standard error stream, 13
standard input, 14
standard input stream, 14
standard output stream, 13
string, 9
string literal, 10
string pool cache, 10

test cases, 2
type cast, 18