Algorithms and Object-Oriented Programming

Class Notes (Grinnell College)

Peter-Michael Osera

January 11, 2019

Contents

I Object-Oriented Programming **1**

1 From C to Java **3**

1.1 Statements and Expressions 3

1.1.1 Expressions 4

1.1.2 Statements 5

1.2 Types 7

1.3 Function Declarations 8

1.4 Procedural Programming 9

1.5 Program Compilation 9

2 Object-Oriented Modeling **13**

2.1 What is an Object? 13

2.1.1 An Example of OO Design: Student 13

2.1.2 The Author and Client Perspective on Objects 14

2.2 The Client Perspective on Objects 14

2.2.1 Object Instantiation 15

2.2.2 Accessing State 16

2.2.3 Invoking Behavior 16

2.3 Blueprints for Objects 17

2.3.1 Anatomy of a Class 17

2.3.2 Class Declaration Syntax 18

2.3.3 Constructors 19

2.3.4 `this` 20

2.3.5 An Example: The Counter Class 20

2.4 Thinking with Objects 21

2.4.1 Abstraction 22

2.4.2 Comments and Style 24

II Reasoning **27**

3 Correctness **29**

3.1 Mental Models of Computation 29

3.1.1 The Substitutive Model of Computation 30

3.1.2 The Stack and Heap Model of Computation 31

3.1.3 Objects and the Heap . . . 34
3.2 Pre-Conditions, Post-Conditions, and Invariants . . . 37
3.2.1 Pre-Conditions and Post-Conditions . . . 38
3.2.2 Small-Scale Reasoning . . . 39
3.3 Invariants and Program Design . . . 43
3.3.1 Iterative Design with Loop Invariants . . . 44

4 Complexity **47**
4.1 Real-world Measurement Versus Mathematical Models . . . 47
4.2 Identifying Relevant Operations . . . 48
4.3 Counting Operations . . . 49
4.3.1 Counting Operations in Loops . . . 50
4.4 Cases of Execution . . . 52
4.5 Big-O Notation . . . 54
4.5.1 Growth of Functions . . . 54
4.5.2 Big-O Notation . . . 55
4.5.3 The Formal Definition of Big-O . . . 55
4.5.4 Additional Asymptotic Bounds . . . 56
4.6 Recurrence Relations . . . 57
4.6.1 An Example: Factorial . . . 57
4.6.2 From Recurrences to Big-O . . . 58

III Structures **61**

5 Algorithmic Design **63**
5.1 Data Organization . . . 63
5.2 Algorithmic Design . . . 63
5.3 Consulting Your Toolbox . . . 64
5.3.1 Abstract Data Types . . . 64
5.3.2 Programming Patterns . . . 65
5.4 Exploring the Problem Space . . . 65

6 Sequential Structures **67**
6.1 The List . . . 67
6.2 Linked- and Array-based Lists . . . 68
6.3 Amortized Analysis . . . 68

7 Polymorphism **71**
7.1 Subtype Polymorphism . . . 71
7.1.1 Static and Dynamic Types . . . 73
7.1.2 Use of the Interface Type . . . 75
7.2 Parametric Polymorphism . . . 76
7.2.1 Autoboxing and Unboxing . . . 79
7.2.2 Generics and Arrays . . . 79

8 Sorting and Searching 83
8.1 Quadratic Sorts . . . 84
8.2 Merge Sort . . . 87
8.3 Quicksort . . . 89
8.4 Generic Constraints . . . 92
8.4.1 Wildcards and Type Bounds* . . . 94

9 Higher-Order Sequential Operations 97
9.1 Iteration . . . 97
9.1.1 Iterator Implementation . . . 99
9.1.2 For-each Loops . . . 101
9.2 Infinite Sequences . . . 102
9.2.1 Streams . . . 102
9.2.2 Stream Processing with Higher-order Functions . . . 103
9.2.3 Functional Interfaces* . . . 106

10 Hierarchical Structures 109
10.1 Hierarchical Structures . . . 109
10.2 The Tree Abstract Data Type . . . 110
10.2.1 Placement of Data . . . 112
10.3 Tree Representation and Operations . . . 112
10.4 Binary Search Trees . . . 114
10.4.1 Tree Traversals . . . 116
10.4.2 Complexity Analysis . . . 117

11 Mapping Structures 121
11.1 An Example: Language Dictionaries . . . 121
11.2 The Map ADT . . . 122
11.3 Association Lists . . . 122
11.4 Tree Maps . . . 124
11.5 Integer Maps . . . 125
11.6 The Set ADT . . . 126

12 Hashing 129
12.1 Revisiting Integer Maps . . . 129
12.1.1 Probing . . . 130
12.1.2 Chaining . . . 131
12.1.3 Removal . . . 132
12.2 Transforming Values to Integers . . . 133
12.2.1 The Hash Function . . . 133
12.2.2 Hash Functions for Primitives . . . 134
12.2.3 Hash Functions for Objects . . . 135

13 Priority Queues **139**
13.1 The Priority Queue ADT 139
13.2 Heaps 140
13.3 The Heap Invariants 141
13.4 Array-based Trees 141
13.5 Heap Operations 142
13.6 Heap Sort 144

IV Large-Scale Program Design **147**

14 Inheritance **149**
14.1 An Example: Shapes 149
14.2 Restrictions on Class Inheritance 153
14.3 Visibility and Dynamic Dispatch 154
14.4 Visibility with Subclasses 154
14.5 Overriding Subclass Behavior 156
14.6 The Mechanics of Dynamic Dispatch 158
14.6.1 Objects on the Heap 158
14.6.2 The Object Class 160
14.7 Is-a Versus Has-a Relationships 162
14.8 The Expression Problem 164
14.9 Case Study: Exceptions 166
14.9.1 Defining Exceptions 166
14.9.2 Handling Exceptions 167

Part I

Object-Oriented Programming

Chapter 1

From C to Java

In order to attract developers, the Java creators gave the language a C/C++-like syntax. While we ultimately have to be mindful that C favors one programming paradigm—procedural programming—and Java favors another—object-oriented programming, we can leverage our knowledge of C syntax to quickly ramp up to Java.

In this chapter, we introduce the procedural programming elements of C programming that translate directly to Java. Recall that *procedural programming* is a programming paradigm where a program is composed of procedural calls, *i.e.*, functions with side-effects. Once we understand how procedural programming is realized in Java, we can better understand what object-oriented programming is and how the two styles of programming contrast and complement each other.

1.1 Statements and Expressions

A programming language is made up different *language constructs* that we can put together in different ways. In C as well as Java, we use two particular sorts of constructs extensively in our programs:

- *Expressions* are programming language constructs that evaluate to a value. A *value* is an expression that can no longer take any more steps of evaluation.
- *Statements* are programming language constructs that carry out one or more side-effects. *Side-effects* are any effects that a program produces that are specifically *not* expressions-evaluating-to-values. These include reading from a file, printing to the console, or mutating a variable.

An example of an expression is an arithmetic expression, *e.g.*, `3 + 5 * (4 - 2)`, which evaluates to the value `13`. An example of a statement is a variable assignment, *e.g.*, `x = 5 + 3`, which has the effect of copying the value `8` into the variable `x`. Note that the variable assignment statement contains an expression, `5+3`, inside of it. The *grammar* of a language specifies the valid ways we can put together a language construct, frequently in terms of other other language constructs that the language provides.

We combine statements and expressions to create programs that do meaningful work. For example, the for-loop:

```
int prod = 1;
for (int i = 1; i <= 10; i++) {
    prod = prod * i;
}
```

repeatedly updates the `prod` variable with the results of the expression `prod * i`. After the loop is done, `prod` effectively contains the result of evaluating `1 * 1 * 2 * ... * 10` or 10!.

1.1.1 Expressions

Java inherits a large part of its expression language from C. Expressions include literals for several types:

- Integers, *e.g.*, `5`,
- Floating-point values, *e.g.*, `3.5`,
- Booleans, *e.g.*, `true`,
- Characters, *e.g.*, `'c'`, and
- Strings, *e.g.*, `"hello world"`.

We can also perform:

- Arithmetic over integer and floating-point expressions with `+`, `-`, `*`, `\textbackslash`, and `\%`.
- Comparisons over integers and floating-point expressions with `>`, `>=`, `<`, `<=`, `==`, and `!=`.
- Boolean arithmetic over booleans with `!`, `&&`, and `||`.
- Bitwise operations over integer values with `&`, `|`, `^`, `~`, `>>`, `<<`, and `>>>`.
- Function calls that produce or *return* values.

Expressions generalize arithmetic expressions, computing by repeated evaluation and substitution. Arithmetic expressions evaluate by repeatedly:

1. Finding the next sub-expression to evaluate by applying *rules of precedence*, *e.g.*, multiplication comes before addition, parenthesized expressions take precedence over non-parenthesized expressions.
2. Evaluating that sub-expression to a value.
3. Substituting the newly acquired value for that sub-expression.

This process continues until the resulting expression is a value. A *value* is an expression that takes no further steps of evaluation. In this context, a number itself is an expression that does not evaluate further, so it is considered a value.

Arithmetic expressions consist of a single type, *e.g.*, integers, and a small set of expression forms, *e.g.*addition and subtraction, that produce integers. Expressions in Java expand on the set of possible types as well as expression forms. Otherwise, they behave exactly like arithmetic expressions with respect to how they compute. For example, here is a more complex expression and its step-by-step evaluation to a final value:

```
    !(5 * 3 > 3 + 2 && !true || false)
--> !(  15  > 3 + 2 && !true || false)
--> !(  15  >   5   && !true || false)
--> !(    true      && !true || false)
--> !(    true      && false || false)
--> !(    true      &&     false     )
--> !(              false            )
--> true
```

Note the rules of precedence at play. For example, with the subexpression `!true || false`, the `!` unary operator has higher precedence than the `||` binary operator, so we evaluate the expression as if it was `(!true) || false` rather than `!(true || false)`.

1.1.2 Statements

Java also inherits much of its statement language from C. In particular, Java features:

- Local variable declaration statements, *e.g.*, `int x;`.
- Variable assignment statements, *e.g.*, `x = 5;`. Note that we can combine variable declaration and initial assignment statements, *e.g.*, `int x = 5;`, and like C, we favor always initializing our locals when they are declared. Java also features the same set of "shortcut" assignment statements as C, *e.g.*, "assign equals" operators such as `+=` and pre- and post-increment and decrement operators `++` and `--`.
- Conditional statements, *e.g.*,

  ```
  if (x < 5) {
      System.out.println("less than five");
  } else {
      System.out.println("not less than five");
  }
  ```

  where the statements constituting the `if`-branch are executed if the *guard* of the conditional (here, `x < 5`) evaluates to `true`. Otherwise, the guard evaluates to `false` and the statements of the `else`-branch are executed. Like C, you can elide the `else` branch or introduce multiple branches using `else if`.
- While-loop statements, *e.g.*,

  ```
  while (x != 0) {
      System.out.println(x);
      x += 1;
  }
  ```

  where the statements that constitute the body of the loop are executed until the guard of the loop evaluates to `false`. Java also has the do-while loop statement where the while-loop body is evaluated once before the guard is evaluated.
- For-loop statements, *e.g.*,

  ```
  for (int i = 0; i < 10; i++) {
    System.out.println(i);
  }
  ```

  which act like while-loops but combine initialization and updating of a variable along with repeated execution of the loop body.
- Switch statements, *e.g.*,

```
switch (x) {
    case 0:
        return 1;
    case 1:
        return 1;
    default:
        return x * 2;
}
```

which performs *case analysis* on a integral value and jumps to the statement after that value's corresponding `case` label. Like C, we have to be careful about case fall-through with cases. The cases are not genuine branches like a conditional; they are merely labels. So unless you use a `break` statement to exit the `switch`, execution flows through subsequent cases.

For example, the following switch statement:

```
switch (x) {
    case 0:
        x = x + 1;
    case 1:
        x = x + 1;
    default:
        x = x + 1;
}
```

If `x` contains the value `0`, then the switch statement will increment `x` three times, eventually storing the value `3` inside of the variable. To fix this problem, the switch-statement should include `break` statements at the end of each case "block" as follows:

```
switch (x) {
    case 0:
        x = x + 1;
        break;
    case 1:
        x = x + 1;
        break;
    default:
        x = x + 1;
        break;
}
```

Unlike expressions, statements do not evaluate to a value. Instead, they produce side-effects on the program. A *side-effect* is any change of state to the program visible to the outside world. For example: The canonical example of this is variable mutation, *e.g.*, `x = 5` changes the value stored at `x` to `5`. But "state" is not restricted only to the mutation of variables. For example:

- Mutating a variable, *e.g.*, `x = 5`, changes the value stored at `x` to `5`.
- Printing text to the console changes the state of the screen.

- Reading data from the network consumes buffered data that has been sent over the wire.
- Invoking a timer requires state to be changed behind the scenes to track how long the timer has elapsed.

In short, anything that is not simply an expression producing a value is a side-effect of one sort or another!

1.2 Types

Like C, Java is a statically-typed language. That is, every expression in our program has a *type* which classifies the value that it produces. This type is known to the compiler before you run a program, and the compiler performs an analysis called *type checking* to ensure that all values in our program are used in a consistent manner. Such programs are called *well-typed*. Programs that do not have this property are called *ill-typed* and produce *type errors* at compilation time.

Types themselves can be divided into two sorts: primitive and compound types. Compound types are made up of other, smaller types. In contrast, primitive types are *atomic*, they cannot be decomposed into smaller types like with compound types. Java shares a number of primitive types with C although the set of values they classify differ slightly:

- `long`: the type of 64-bit (8 byte) signed integers.
- `int`: the type of 32-bit (4 byte) signed integers.
- `short`: the type of 16-bit (2 byte) signed integers.
- `byte`: the type of 8-bit (1 byte) signed integers.
- `double`: the type of 64-bit (8 byte) floating point values.
- `float`: the type of 32-bit (4 byte) floating point values.
- `boolean`: the type of boolean values, *i.e.*, `true` and `false`.
- `char`: the type of single 16-bit Unicode characters.

Notably, Java has a richer type system than C in the sense that a `boolean` is not simply an integer; a `boolean` is its own distinguished type that cannot be interchanged with integers. In particular, in C, the expression `1 + true` is well-typed (although its resulting value is undefined) whereas in Java, the expression is ill-typed and would be rejected at compile time.

In C, aggregate types includes arrays, pointers (which generalize arrays), and `struct`s. Java differs from C with respect to all three of these language features:

- In Java, arrays are their own type distinct from pointers, written `T[]` for an array of `T`s. To create an array, we use a `new` array expression which creates an array on the heap of the specified size. The syntax of the `new` array expression is: `new <type>[<size>]`. For example the variable declaration `int[] arr = new int[100]` declares a variable called `arr` that references an array (of `int`s) of size 100 on the heap.

  Indexing into an array in Java is identical to C with an *array indexing expression*, *e.g.*, `arr[4]` to retrieve the element at index 4 (the fifth element) of the array. Unlike C, Java performs *bounds checking* to ensure that the index provided to an array indexing expression is in range. If an invalid index is accessed of an array, rather than performing undefined behavior, a Java program will throw an *exception* signifying that an erroneous condition has occurred. Consequently, this means that a

Java array must store its length in addition to its element to perform this run-time check. We can access the length of an array directly by using a *field access expression* to access the `length` field of the array. Combining all this together, the following code snippet sums up the elements of an integer array called `arr`:

```
int[] arr = new int[100];
/* Load arr with some values... */
int sum = 0;
for (int i = 0; i < arr.length; i++) {
    sum += arr[i];
}
```

- Java does not explicitly have a pointer type although certain types in Java that we will discuss later (so-called "reference types") are always implicitly handled through pointer.
- Java does not have a `struct` construct. It instead has a `class` construct which you can think of as a `struct` with additional features. `class` declarations are the cornerstone of Java programming in an object-oriented style, so we'll defer our discussion of classes to later.

1.3 Function Declarations

The basic programming-in-the-small constructs of Java are very similar to C. However, the way that we organize our Java programs differs significantly from C programs. Recall that the canonical "Hello World!" program in C is written as:

```
// In hello.c
int main(void) {
    printf("Hello World!\n");
    return 0;
}
```

In contrast, the equivalent Java program is written as:

```
// In Hello.java
public class Hello {
    public static void main(String[] args) {
        System.out.printf("Hello World!\n");
    }
}
```

Note the differences:

- The "function" `main` is wrapped in a *public class declaration* called `Hello`. The name of the file, `Hello.java`, is the same as the class name, `Hello`.
- Function declarations are prepended with the modifiers `public static`.
- The function signature for the `main` function in Java is `void main(String[])`. That is, `main` must be a function that takes an array of strings (the command-line arguments to the program) and returns nothing.

- To print text to the console, we use `System.out.printf` rather than simply `printf`. Alternatively, and more standard, we can also use `System.out.println` which prints a string to the console and also adds a newline character (`'\n '`) to the end of the string.

All of these additional pieces of syntax—the class declaration, the `public static` modifiers—have important meaning to our program. However, they do not directly impact our ability to write basic procedural programs in Java, so we'll defer discussion of them to later chapters. For now, we will write our Java programs as follows:

1. All functions are wrapped in a public class declaration whose name is the same as the source filename (minus the `.java` part).
2. All functions are prepended with `public static`.

Here is a basic template for you to utilize for your starting Java programs:

```
// In MyProg.java
public class MyProg {
    public static void main(String[] args) {
        // My main function code goes here
    }
}
```

1.4 Procedural Programming

With all of these elements—expressions, statements, and functions—we can adapt our *procedural programming* model of programming in C to Java. Procedural programming is a programming paradigm where our programs are composed of procedures—functions with side-effects. If our program solves a problem, we *decompose* that problem into smaller problems to solve, reflecting this decomposition in the functions that we author. For example, the program `SumArgs` found in section 1.4 prints the sum of the command-line arguments given to the program.

The program decomposes the problem of printing the sum of the command-line arguments by first transforming the arguments from an array of strings to an array of integers (`transformArgs`) and then summing up the elements of an array of integers (`sum`). This decomposition is reflected in `main` by the order of the calls to the helper functions. In our procedural style, we ideally want our `main` functions to directly reflect our high-level strategy for solving the problem at hand.

Even though we will ultimately program in an object-oriented style in Java, we will still use these procedural decomposition techniques to structure how our top-level `main` function looks and behaves.

1.5 Program Compilation

We used the GNU Compiler Collection (`gcc`) to compile our C programs to executable programs. Recall that *compilation* is the process of translating a program between different forms, in this case, from textual source code to executable machine code. To compile our Java programs, we invoke the Java compiler, `javac`, from the command-line similarly to `gcc`. If the `Hello.java` source file is in the current directory, we can compile it using `javac` as follows:

```java
public class SumArgs {
    public static int[] transformArgs(String[] args) {
        int[] ret = new int[args.length];
        for (int i = 0; i < args.length; i++) {
            ret[i] = Integer.parseInt(args[i]);
        }
        return ret;
    }

    public static int sum(int[] arr) {
        int ret = 0;
        for (int i = 0; i < arr.length; i++) {
            ret += arr[i];
        }
        return ret;
    }

    public static void main(String[] args) {
        System.out.println(sum(transformArgs(args)));
    }
}
```

Figure 1.1: Sample procedural program that sums up the command-line arguments given to the program. To pass command-line arguments to a Java program, you specify them after the class name when invoking `java`, *e.g.*, `java SumArgs 1 2 3`.

```
$> ls
Hello.java
$> javac Hello.java
$> ls
Hello.class Hello.java
```

However, `javac` does not produce a program that we can run directly! Note that the output of compiling `Hello.java` is a *class file* called `Hello.class`. Class files are compiled Java programs in an intermediate instruction set called *Java bytecode*. These class files are not directly executable; instead, we must use an *interpreter program*, `java`, which executes the program contained in the class file.

```
$> java Hello
Hello World!
```

Why does Java produce class files rather than executable machine code directly? Java bytecodes are a *machine-independent* representation of a Java program. This allows us to compile a Java program once locally and then distribute the resulting class files that constitute our program to anyone, irrespective of the operating system they are running. As long as they have an appropriate version of the Java runtime on their machine (which contains the `java` program), they can execute our program without any additional steps of compilation or configuration!

Exercise (Compilation Pipeline) Write a Java program in a file called `MyGreeting.java` that prints out a modified greeting of the form `"Hello, from <your name here>"`! Use this simple program to ensure that you understand how a Java program is organized and that you can compile a program from the command-line properly. Make sure that you save this program as a template for future Java programs that you write!

Chapter 2

Object-Oriented Modeling

Previously, we studied programming-in-the-small in Java, mapping our knowledge of C into this new, C-like language. This week, we will explore *object-oriented programming* in Java and how this style of programming is fundamentally different from writing programs in C.

2.1 What is an Object?

You may have heard the term "object" used in Racket or C as a synonym for "data". However in an object-oriented programming language like Java, *object* refers to a very specific kind of program entity:

Definition 1. *An object possess* state *and* behavior.

So far we have seen two types of objects in Java:

Strings Its state is the (immutable) sequence of characters it contains. An example of its behavior is the `trim()` method which produces a new string that is identical to the old string except the whitespace on either end is removed.

Arrays Its state is the sequence of elements it contains (accessed with index notation) along with its length (accessed via dot notation, *i.e.*, `arr.length`). It has no associated behavior.

In Java, we realize the state of an object as *fields* or *instance variables* of an object. Recall that a C `struct` is a collection of a data fields. The instance variables of an object are analogous to these data fields of a `struct`.

We realize the behavior of an object with *methods.* A method is a function that we "call on" a particular object. For example with the `trim()` method above, if we have a variable `s` of type `String`, then we can call its `trim` method as follows: `s.trim()`.

2.1.1 An Example of OO Design: Student

When we design a program in an object-oriented style, we think about modeling the data of our program as a collection of objects that interact with each other through their methods. For example, in a program that manages a course registration database, we will likely need to model a student. In this context, what are the *state* and *behavior* of a student? Here are some examples:

State First name, last name, birthday, gender, age.

Behavior Register (for a course), drop (from a course), withdraw (from the semester).

And we can consider other pieces of state and behavior as our problem demands it. The state of the student becomes fields or instance variables of our student objects. The behavior becomes methods, *e.g.*, we might realize the register behavior as the method `void register(Course c)` which takes a `Course` object (yet to be defined) as input, registers the student for the course, and returns nothing. But which student is registered for the course? The student that we *call the method on*, for example:

```
Student s = /* Some initialization... */
Course csc207 = /* Some more initialization... */
s.register(csc207);
```

We use *dot notation* to invoke a method on an object, just like how we use dot notation to access a field of an object, *e.g.*, the `length` field of an array. Unlike field access, however, we do not provide only the name of the method of interest, we also provide the list of arguments to pass to the method. In effect, the student `s` is the *implicit* first argument to the method—it is neither specified in the method declaration nor provided in list of arguments to the method.

Note that we have not chosen all the possible pieces of state and behavior for our student objects. For example, a student certainly has a height. However we likely do not need to know a person's height to manage their course registration. This is the fundamental *design* question of object-oriented programming: how do we best and most concisely represent the salient features of the data in our program? To answer this question, we must take into account the context in which we are designing the object, what types best capture our requirements, and whether we can be clever about our representation to avoid unneeded complexity.

2.1.2 The Author and Client Perspective on Objects

Another way to look at objects is from two dual perspectives. Someone is responsible for designing the objects we use, *i.e.*, deciding what the state and behavior of that object is. We call this party the *author* of the object. Once we have a definition of an object, people can then use those objects in their own program. We'll call these people *clients* of the object.

These two perspectives on objects are especially important for understanding the relationship between author and client when *interfaces* are involved. There are potential obligations on either side of the object line—author or client—that must be met to guarantee that the program works correctly. We'll discuss these various sorts of *pre-conditions*, *post-conditions*, and *invariants* in the next chapter.

For now, this division is important for our purposes because it outlines our trajectory for discussing objects. We'll first talk about using objects as clients: creating already-defined objects, accessing their state, and invoking their behavior. All of this will feel eerily like using `struct`s in C. Then we'll talk about all the details of authoring objects—defining templates for objects called *classes* as well as all of the features we can throw into our objects to make them robust abstractions for others to use.

2.2 The Client Perspective on Objects

As a client of objects, there are three fundamental operations we can perform on them:

1. We create or *instantiate* objects.
2. We access the state or *instance variables* of objects.

3. We invoke the behavior or *methods* of objects.

We'll explore how we do each of these three operations in Java.

2.2.1 Object Instantiation

For the two objects that we've seen so far, arrays and strings, we saw special syntax for their creation. For arrays, we could either:

1. Instantiate an array of a given size and type whose elements are initialized to be the default values of that type or
2. Instantiate an array to an initial set of contents with special *array initialization syntax*.

```
int[] arr1 = new int[20];                     // initialized to an array of 20 zeros.
int[] arr2 = new int[] { 0, 1, 2, 3, 4, 5 };  // initialized to an array containing 0--5
```

For strings, we used string literals to create strings, *e.g.*, `"Hello World!"`. However, we have already learned of an alternative way to create a string. We can create a string from an array of characters as follows:

```
char[] arr = new char[] { 'h', 'e', 'l', 'l', 'o' };    // Note: no null character!
String s = new String(arr);                             // The string "hello"
```

This is an example of the primary way of creating objects: *new expressions*. In general, a new expression has the following form:

```
new <class name>( <parameters> )
```

This syntax invokes the *constructor* of an object which:

1. Allocates the memory for that object.
2. Initializes that object given the arguments to the constructor.

Here, the expression `new String(arr)` creates a new string object by invoking the `String` constructor that takes a `char[]` as input. The result is a string containing the characters found in that array.

Another example of object instantiation that we've seen is the `Scanner`:

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

`new Scanner(System.in)` creates a new `Scanner` object by invoking the constructor that takes an `InputStream` object as input. (It turns out that `System.in` is an object of type `InputStream`. More specifically, `in` is the object, and it is a *static* member of the `System` class, an important distinction we'll discuss shortly.)

2.2.2 Accessing State

To access the state of an object, we use *dot notation*. It has the following syntax:

```
<object>.<field name>
```

An example of this syntax is accessing the length of an array:

```
int[] arr = new int[5];
System.out.println(arr.length);
```

Think of the dot ('.') as a *binary* operation like addition, except that the left-hand side of the dot is an expression that must evaluate to an object and the right-hand side is the name of the field of the object that we would like to access.

Normally, we would be able to *mutate* or change a field that we have access to, just like how we can modify the fields of `struct`s in C. However, if we try to do this to an array we get the following error:

```
arr.length = 6
/* ERROR: cannot assign a value to final variable length */
```

This is because the `length` field of the array is constant or `final`; it cannot be changed.

2.2.3 Invoking Behavior

Recall that the behavior of an object corresponds to the set of *methods* that the object exposes to a client. To invoke a method on an object, we also use dot notation:

```
<object>.<method name>( <parameters> )
```

For example, we used the `charAt` method of String objects extensively in previous labs and homework:

```
String isbn = "123456789X";
System.out.println(isbn.charAt(9));   // 'X'
```

`s.charAt(n)` fetches the `n`th character from the string `s`. Here, the object is `s`, a string, and the name of the method is `charAt`. The single parameter to `charAt` is the index of the `char` we wish to fetch.

Method invocation looks a lot like function calls and indeed they are very similar. If we're not being ultra-precise about our language, we may call `charAt` a function call or function call-looking thing a method invocation, and that is fine when discussing code casually. However, let's make the distinction between the two explicit, especially since we are coming from a C background:

1. A *method invocation* is always invoked on a particular instance of an object, namely, the object that appears to the left of the dot.
2. A *function call* or *application* is not invoked on a particular object. It looks like the normal C function calls that you are used to.

As we transition into the world of objects, it'll be important to keep the two types of function-like calls straight in our head.

Exercise (String Methods) In a class called StringMethods, write a `static` method called `endsWithRep` with method signature:

```
public static boolean endsWithRep(String s1, String s2, int n);
```

that returns `true` if `s1` ends with `n` repetitions of `s2`.

Here are some example invocations of `endsWithRep`:

```
endsWithRep("foobarbar", "bar", 2);    /* returns true  */
endsWithRep("foobarbar", "baz", 1);    /* returns false */
```

To write this method, you can use string concatenation (the + operator) to build up a string that is `n` repetitions of `s2` and the `boolean endsWith(String suffix)` method of the `String` class which returns `true` if `suffix` is indeed a `suffix` of the string the method is called on. In the `main` method of `StringMethods`, demonstrate that your method works by printing the results of the two examples above to the console.

2.3 Blueprints for Objects

Now that we have seen how to interact with objects from the client perspective, let's now discuss how to specify objects. To do this, we'll formally introduce the *class* construct that has been in every one of our Java programs so far.

2.3.1 Anatomy of a Class

Recall that an object is a programming entity that contains state and behavior. To specify what kinds of state and behavior an object contains, we use *classes*. We say that an object is an *instance* of some class. For example, recall from the previous section we defined a Student and defined the following pieces of state and behavior for it:

State First name, last name, and age.

Behavior Register, drop, and withdraw.

Here is how we would take these pieces of state and behavior and define them in a Java class:

```
public class Student {
    public String firstName;
    public String lastName;
    public int age;

    public Student (String firstName, String lastName, int age) {
        /* ... */
    }

    public boolean register(String course) {
        /* ... */
    }

    public void drop(String course) {
        /* ... */
    }

    public void withdraw() {
        /* ... */
    }
}
```

For the time being, we've elided the implementations of the constructor and methods. But regardless, we can use this class as follows:

```
Student s = new Student("Ada", "Lovelace", 23);
s.register("csc 207");
```

The state of an object translates into *field* or *instance variable declarations* in our class. You should recognize these from C; these look and behave the like the field declarations of a `struct` definition. The behavior of an object translates into *method* definitions. These are like function declarations but appear within the class definition *without the `static` modifier*. Adding `static` no longer associates the function with an instance of the class—a distinction that made the function a method—but rather with the class itself. We'll explore this distinction in more detail later as it is one of the greatest points of confusion for students transitioning from the programming-in-the-small world of C to the object-oriented world of Java.

The `public` annotations on the class, the fields, and the methods determine the *visibility* of that particular program entity. Keeping the client versus author distinction in mind, an entity marked `public` is usable to everyone—clients and author alike. In contrast, a entity marked `private` is usable only by the author. This is useful for hiding fields and methods that concern the *implementation* of the class, *e.g.*, auxiliary functions and state that we don't want the outside world to know about.

A class is a sort of swiss-army knife in object-oriented languages like Java; it does a bunch of stuff:

- Classes act as a blueprints for objects as we discussed before.
- Classes are the way of defining *user-defined* types in Java.
- Classes act as a *namespace* for collections of `static` functions and variables.
- Classes act as an abstraction mechanism separating *features* from *implementation* through *interfaces*.

It is easy to conflate all of these features together, especially if Java is your first language. However, with multiple languages under our fingertips, we can see that all of these features serve their own distinct purposes.

2.3.2 Class Declaration Syntax

Class declarations take on the following form:

```
<visibility modifier> class <class name> {
    <field and method declarations>
}
```

A program in Java is defined to be a collection of class declarations. The class declaration may optionally be preceded by a visibility modifier, *e.g.*, `public` or `private`. The visibility modifier may be left out which turns out to have a different meaning from `public` or `private` ("package"-protected which we'll discuss later when we talk about Java's package system).

A class definition contains a number of *declarations*:

1. Field declarations.
2. Method declarations.
3. Constructor declarations.

Field declarations look like local variable declarations but exist outside of any particular method but inside a class definition:

```
<visibility modifier> <type> <name>;
```

For example, `public String name;` in the declaration of `Student` above declares a field of type `String` named `name`. Every instance of a `Student` has their own `name` field.

Method declarations look a lot like the function declarations we have seen so far:

```
<visibility modifier> <type> <name>( <arguments> ) {
    <statements>
}
```

except that there is no `static` in the signature of the method. As discussed ealrier, this is the distinction between a *method*, a function tied to a particular object, and a *static function*, a function not tied to a particular object but the overall class.

2.3.3 Constructors

Recall that fields and methods gives us way of using objects. Constructors give us ways of creating objects, a process called *instantiation*. The constructor defines how we should *initialize* a freshly-created object of the given class. We define a constructor in a class similarly to how we define a method except:

1. The name of the method is the name of the class.
2. There is no return type.

For example, recalling the constructor of our `Student` class above:

```
public Student (String firstName, String lastName, int age) {
    /* ... */
}
```

This is a constructor that takes three arguments: two strings corresponding to the person's name as well as their age. This constructor allows us to create a `Student` using a `new` expression as follows: `new Student("Ada", "Lovelace", 23)`.

2.3.4 `this`

In the example above, we have omitted the implementation of the constructor. What should the constructor do to initialize a new `Student`? A sensible sketch of an approach is:

```
public Student (String firstName, String lastName, int age) {
    // Initialize the firstName field of Student with the parameter firstName
    // Initialize the lastName field of Student with the parameter lastName
    // Initialize the age field of Student with the parameter age
}
```

However, how do we access the fields of the object that we are creating? Recall that the syntax of a field access is `<object>.<field name>`, but what goes to the left-hand side of the dot? If we need to refer to the object that we are currently instantiating (in a constructor) or called the method on, we use the `this` keyword:

```
public Student (String firstName, String lastName, int age) {
    this.firstName = firstName;
    this.lastName  = lastName;
    this.age       = age;
}
```

In a method or constructor, `this` is an expression that evaluates to the object that is the subject of the constructor or method call.

2.3.5 An Example: The Counter Class

As a complete example to study, let's consider creating a class that represents a simple counter that we can increment. What is the state and behavior of a counter?

State The current value of the counter, an integer.

Behavior Incrementing the counter.

Now let's translate this into a simple Java implementation:

```
public class Counter {
    public int value;

    public Counter() {
        this.value = 0;
    }

    public void increment() {
        this.value += 1;
    }
}
```

Here's an example of using this `Counter` class:

```
Counter c1 = new Counter();
Counter c2 = new Counter();
System.out.println(c1.value);   // 0
System.out.println(c2.value);   // 0
c2.increment();
c1.increment();
c1.increment();
c2.increment();
c2.increment();
System.out.println(c1.value);   // 2
System.out.println(c2.value);   // 3
```

Note that each instance of the `Counter` possesses a distinct `value` field. So each call to `increment()` increments the `value` field of the counter that the method is called on.

A final note: because we annotated `value` with the `public` visibility modifier, anyone can change the value of a counter. For example:

```
c1.value = 5;
System.out.println(c1.value);   // 5
System.out.println(c2.value);   // 3
```

This may be fine for our simple purposes, but we may want to *hide* this field so that non-counter code cannot change the value directly. We can accomplish this by marking the field `private` instead of `public` although there are design considerations to make in choosing one modifier over the other. We'll discuss these considerations in more detail shortly.

Exercise (An Example Class: Dogs) Define a class called `Dog` in an appropriately named Java file that defines a class that represents dogs. Define your `Dog` so that it has at least three fields, a constructor, and a method. You may define whatever fields and methods for your `Dog` class that you would like. (I didn't give you any additional parameters so you can design your class in whatever context you desire). If you are at a loss for creativity, one recommendation is to define your method to be a `bark` method which returns nothing and makes the dog bark the value of its properties to the console.

2.4 Thinking with Objects

In Java, we decompose our problems not in terms of mathematical functions—a *functional* style—as in Racket, not in terms of procedures (functions with side-effects)—a *procedural* style—as in C but in terms of objects—an *object-oriented style*. At a first glance, the differences between functional or procedural and the object-oriented programming seem insignificant. We still have to think about both data and functions in either a functional or procedural style, and we frequently reason about them together. However, in Java, we unite data and methods together under the all-encompassing object construct. This simple change in code organization fundamentally changes the way that we approach program design in Java.

The classes that we identify, design, and implement in order to solve our problem become little packages of code. Ideally, these little packages satisfy a few properties:

1. They are *small and as simple as possible*. It would defeat the purpose of decomposing a problem into classes if the classes were as complex as the original problem!

2. Related, they are *limited in scope*, ideally, serving only a single distinct purpose in the overall program.
3. They function as *independent units* (as much as possible). This allows us to reason about the packages for correctness independently, either during debugging or testing.
4. They possess a *well-defined* interface with clear guarantees about inputs and outputs (when appropriate).
5. We are able to *hide* the details of the package that are unnecessary for clients of the package to know about, *i.e.*, its implementation details.

The process of bundling data and behavior together into packages that satisfy these properties is called *encapsulation*. Java allows us to accomplish this with the `class` construct. However, classes alone only allow us to bundle code; it is up to us to enforce these properties through good object-oriented design principles.

2.4.1 Abstraction

The first three properties imply that our classes should be kept small and specific in their purpose. The final two properties deal with *abstraction*, the hiding of a system's implementation through an interface. For example, consider the `Counter` class we've used as our running example so far:

```
public class Counter {
    public int value;

    public Counter() {
        this.value = 0;
    }

    public void increment() {
        this.value +=1 ;
    }
}
```

The interface that a class specifies (for one of its instances) contains all of the (accessible) fields and methods of that class. For example, a `Counter` exposes:

- A way to construct an instance via a no-argument constructor,
- A `value` field that is the current value of the counter, and
- An `increment()` method to increment the counter.

These three things constitute a user's *interface* to a counter. One suspicious design decision here is that we have exposed the `value` field to the user by marking it `public` This may be undesirable because a user can set the value of a counter to any value that they want, *e.g.*,

```
Counter counter = new Counter();
counter.value = -42;
```

If we wished to restrict the counter from ever going negative or more specifically, only allow a user to change the value of the counter through `increment()` then this choice of interface is not sufficient. However, we can't simply remove value from the class because it needs it to keep track of the number of calls to increment()! We need some mechanism to *hide* value from clients but still keep it around so that a counter can use it internally.

In Java, we accomplish this sort of hiding of members with *privacy modifiers*. So far, we have seen the public privacy modifier which makes a member visible to everyone. We fix this problem by modifying the value field with the private modifier which makes a member visible to only the class containing the member.

```
public class Counter {
    private int value;

    public Counter() {
        this.value = 0;
    }

    public void increment() {
        this.value +=1 ;
    }
}
```

Now, with the value field marked as private, clients of the class can no longer access it. In particular, the code above that changes the counter's value to -42 produces the following compiler error:

```
Counter.java:16: error: value has private access in Counter
        c.value = -42;
         ^
1 error
```

This is what we want! But there is one problem: we cannot access value at all! For example, if we wanted to print out value:

```
System.out.println(counter.value);
```

We receive the same error.

So how do we fix this problem? We create an alternative route to access value: a public method that simply returns value:

```
public class Counter {
    private int value;

    public Counter() {
        this.value = 0;
    }

    public void increment() {
        this.value +=1 ;
    }
    public int getValue() {
        return value;
    }
}
```

Now, we can use `getValue()` to retrieve `value` without exposing a way to change it via assignment.

```
System.out.println(counter.getValue());
```

Such a kind of method that simply returns the value of a field is so commonplace in Java that we have a special name for it: a *getter method*. If we also wanted to enable a client to change a field, we could create a corresponding *setter method*:

```
// In Counter...
public void setValue(int value) {
    this.value = value;
}
```

A getter and setter method combined provides all of the functionality that a `public` field provides but does not actually expose the field itself. So why would we want to create both a getter and setter? There may be an `invariant` of the counter—a property—that we would like to preserve, *e.g.*, that the counter should never be negative:

```
// In Counter...
public void setValue(int value) {
    if (value >= 0) {
        this.value = value;
    } else {
        throw new IllegalArgumentException();
    }
}
```

We can use `setValue` to enforce this property. If the user provides an inappropriate argument, then we signal an error with an exception. We'll discuss these mechanics and design considerations when we talk about interfaces in the next chapter.

2.4.2 Comments and Style

When designing abstractions, we rely on our type system (when available) to enforce those abstractions. Take a look at the signature of `setValue(value)` again:

```
public void setValue(int value) { /* ... */ }
```

Java enforces that we must call `setValue` with exactly one argument and that argument must be an `int`. However, the signature alone does not tell the user that they are not allowed to provide a non-negative argument. The exception we threw in the implementation signals to the user that they messed up, but at runtime. Ideally we would like to catch these sorts of errors at compile time, but we have no way of enforcing these properties with Java. Instead we must resort to documenting them with comments.

Java provides excellent facilities for commenting code: *Javadocs*. Here is an example of using the Javadoc facilities in Java:

```
/**
 * Sets the value of the counter.  This value must be non-negative.
 *
 * @param value the new, non-negative value the counter.
 * @throws IllegalArgumentException if a non-negative value is given.
 */
public void setValue(int value) { /* ... */ }
```

Javadocs are special comments above declarations of program elements (*e.g.*, classes or methods). They start with `/**` and end with `*/`. They include special *tags* for various parts of the documentation. The most important of these for methods are:

- `@param <name> <description>`: Used to document a method parameter.
- `@return <description>`: Used to document a return value.
- `@throws <name> <description>`: Used to document an exception the method may throw.

When building rich abstractions, comments become a necessary tool to ensure people know how to use your code and what to expect from it!

Exercise (The Student Class Revisited) Fix the version of the `Student` class below so that it (a) does not expose its fields directly and (b) has appropriate Javadoc comments. Your updated class should use privacy modifiers and setter and getter methods to expose read/write access to fields as necessasry. Your Javadoc comments should contain tags for the return values and parameters of any methods or constructors that you document.

```
public class Student {
    public String firstName;
    public String lastName;
    public int id;
    public int age;

    public Student (String firstName, String lastName, int id, int age) {
        this.firstName = firstName;
        this.lastName  = lastName;
        this.id        = id;
        this.age       = age;
    }
}
```

Part II

Reasoning

Chapter 3

Correctness

One of the most appealing aspects of computer programs is that they have a close connection to mathematics, in particular, logic. We use these connections implicitly every day when designing our programs. However, it is worthwhile to deeply understand them so that we can use them to formally reason about our programs and their behavior.

When applying formal reasoning to computer programs, we are interested in determining if a *property* holds of a particular program. There are many sorts of properties about programs that we care to analyze; the two we'll focus on in this course are:

- *Correctness*: does the program produce the desired output?
- *Complexity*: how many resources does the program consume?

Correctness should be important to everyone that writes programs as we care deeply that our program accomplishes the task we set out to do! Previously, you checked the correctness of your program by *testing* it, *i.e.*, running the program on particular inputs and ensured that it produced the output that you expected. Testing is a powerful methodology for ensuring program correctness but it has its downsides. In particular:

- In certain situations, *e.g.*, programs that control rocket ships or automated cars, the cost of testing is prohibitive either in terms of money (the rocket can blow up) or risk (the car could hit someone).
- When developing a large-scale application, we may develop a small piece of it and want to have some assurances that we have done the right thing, but the application is not close to a state in which we can run.
- We would like guidance on how to write our programs beyond simply knowing correlations between inputs and outputs.

As a complement to testing, we can use *mental models of computation* to reason about the behavior of our programs for all possible inputs and without running them explicitly.

3.1 Mental Models of Computation

One view of mathematics is that it is the study of abstractions for the purpose of modeling real-world phenomena. With these models, we can compare, categorize, and predict the behavior of these phenomenon. However, because these models are formed through observations of the phenomenon, there is the potential for disconnects between the models and reality, *e.g.*, in physics, classical mechanics breaks down in the presence of objects approaching the speed of light. In contrast, the behavior of programming languages are usually designed with a mathematical model of computation in mind. Thus there is no disconnect between how a programming language behaves and our own mental model operates, as long as we have the correct model in mind!

A Note on Paper To practice the skill of programming, we write software artifacts, and we stress that the process of developing these artifacts is invaluable in your development as a computer scientist. Likewise, to practice mathematics, we write stuff down on paper, whether that is diagrams, mathematical definitions, or proofs. Treat any mathematical paper exercise like programming by actually doing said exercise in full detail: pull out paper and write stuff down! It turns out that resorting to paper is also a great way to work out a programming problem, so it's a good habit to get into now if you are not doing it already!

3.1.1 The Substitutive Model of Computation

Recall that in most programming languages, there are two kinds of program fragments:

1. Expressions which evaluate to a value.
2. Statements that do not evaluate to a value but do some meaningful work.

Let's focus on building a model for this first class of program fragments. You are already familiar with this model of computation: it's what you do when you evaluate mathematical expressions. For example, evaluation of the expression `12 + 6 * (4 - 2)` proceeds as follows:

```
    12 + 6 * (4 - 2)
--> 12 + 6 *    2
--> 12 +    12
-->     24
```

Each line represents one step of evaluation we take to evaluate the expression. The final result, `24`, is a value: an expression for which we can no longer take any evaluation steps.

To find and carry out the next step of evaluation, we do the following:

1. Find the next sub-expression to evaluate based off of precedence rules.
2. Carry out the evaluation of that sub-expression to a value.
3. Substitute the value for that sub-expression.

Finally, we repeat this process until we arrive at a final value.

With programming language expressions, the process is identical except that we extend this sort of reasoning to other sorts of expressions, not just arithmetic ones. For example, using the rules of precedence, we know that when evaluating the expression `arr.length > i - 1`, we know that `-` has higher precedence than `>`, so we evaluate `i - 1` first rather than `arr.length > i`. If we did evaluate `arr.length > i`, we would encounter a type error because the result of `arr.lenght > i` is a boolean, but then we try to subtract one from it.

Functions only add a slight complication to this model. For now, we'll only consider pure functions, *i.e.*, functions that only return a value and do not produce any side-effects like mutating (re-assigning) global variables or printing to the console. An example of a pure function is this simple increment function:

```
public static int inc(int x) {
    return x + 1;
}
```

When evaluating an expression containing a function call, we follow the steps above as normal. However, when we evaluate a function, we do the following:

1. Evaluate the arguments to the function call down to values.
2. Substitute the actual values passed to the function for the formal parameters (*i.e.*, variables) of the function.
3. Evaluate the body of the function to a final value.
4. Substitute this resulting value for the function call.

As an second example, consider evaluating the expression, `inc(1) + inc(5*2)`:

```
    inc(1)  + inc(5*2)
--> (1 + 1) + inc(5*2)
-->    2    + inc(5*2)
-->    2    + inc(10)
-->    2    + (10 + 1)
-->    2    +    11
-->        13
```

3.1.2 The Stack and Heap Model of Computation

The substitutive model of computation is a simple model to work with. And indeed, we can use it to reason about our programs when there are no side-effects (which is one of the reasons why we try to minimize the number of variables and assignments in our code). However, we inevitably need to deal with side-effects whether it is dealing with variable that change values, printing text to the console, or reading input from the user. The substitutive model does not deal with side-effects as it deals exclusively with expressions instead of statements whose purpose is to produce side-effects.

We therefore introduce another model of computation, the stack and heap model, which will allow us to reason about side-effects, in particular, variables whose values change over time. Recall that *the stack* contains *activation records* of each currently active function call in the program. These records contain the parameters and local variables for that function. A stack is an appropriate data structure for capturing function calls because function calls obey a first-in-first-out (FIFO) discipline:

- The most recently called function is always the currently active function call.
- When we return from a function, we always return from the most recently called function first.

The heap on the other hand is where all of our dynamically allocated memory resides. In C, we dynamically allocated memory when ever we called `malloc`. In Java, all non-primitive data is allocated on the heap—the most critical difference between Java and C is that you don't have control of where your data resides in memory!

How does the stack and the heap model work? Let's demonstrate it with a step-by-step example. Consider the following code snippet:

```
public class Program {
    public int inc(int x) {
        int y = x + 1;
        return y;
    }
    public static void main(String[] args) {
        int x = 0;
        System.out.println(x);
        System.out.println(inc(x));
        x = inc(x+1);
        System.out.println(x);
        System.out.println(inc(x));
    }
}
```

First, read the code and try to guess what the output of the program will be. Now, we'll verify your hypothesis by tracing through the execution of the program using the stack and heap model of evaluation. We first start by calling the `main` method which takes as a parameter an array of strings collected from the command-line invocation of the program. Our stack and heap at the top of start look as follows:

```
    Stack         Heap
    -----         ----
  --main
  args [ o-]----->[]
```

The stack contains a single frame recording that we are in `main`. The one parameter to `main`, `args`, is a reference to an empty array.

We then begin executing the statements of `main` line by line. The next line is a variable declaration. We add this entry to the stack:

```
    Stack         Heap
    -----         ----
  --main
  args [ o-]----->[]
     x [ 0 ]
```

Because `x` has a primitive type, `int`, the variable on the stack holds the integer value directly rather than holding a reference to a value stored on the heap. On the next line, we print out the current value of `x` which is `0`:

```
    Stack         Heap       Output
    -----         ----       ------
  --main                     0
  args [ o-]----->[]
     x [ 0 ]
```

Next, we call `inc` passing in the current value of `x` which is `0`. Calling `inc` creates a new frame on the stack which contains the lone parameter for `x`. This parameter is loaded with the value of `x`. Note that the `x`s refer to distinct memory locations here.

```
    Stack        Heap       Output
    -----        ----       ------
  --main                    0
  args [ o-]----->[]
     x [ 0 ]
  --inc
     x [ 0 ]
```

Inside the function, we declare a local variable `y`, adding it to `inc`'s stack frame. `y` is initialized to be one greater than `x`. Again, it is the local `x` value in `inc` that we use rather than `main`'s `x` because `inc` is the currently active function.

```
    Stack        Heap       Output
    -----        ----       ------
  --main                    0
  args [ o-]----->[]
     x [ 0 ]
  --inc
     y [ 1 ]
```

The function ends by returning the current value of `y` which is `1`. When we return, the stack frame for `inc` is deleted, leaving the stack frame for `main`, and we continue execution of the program where we left off in `main`. Returning from a function works just like the substitutive model: we substitute the result of the function call for the function call itself and continue evaluating the expression. This means that we print out `1` next.

```
    Stack        Heap       Output
    -----        ----       ------
  --main                    0
  args [ o-]----->[]        1
     x [ 0 ]
```

Next, we re-assign the value of `x`. To do this, we call `inc` again passing in `x + 1`. Note that the value of `x` in `main` did not change as a result of the previous function call! This is because `inc` only changes the value of its local variables; it is unable to affect the local variable of `main` directly.

Therefore, when we call `inc` a second time, we pass `1`. Walking through the execution of the function, we return from `inc` in the following state:

```
    Stack        Heap       Output
    -----        ----       ------
  --main                    0
  args [ o-]----->[]        1
     x [ 0 ]
  --inc
     x [ 1 ]
     y [ 2 ]
```

From this, we see that we return `2` from the call to `inc`. This value is then stored in `x` (in `main`) as the result of the assignment:

```
    Stack         Heap        Output
    -----         ----        ------
  --main                      0
  args [ o-]----->[]          1
     x [ 2 ]
```

We then we print out the current value of x which is 2.

```
    Stack         Heap        Output
    -----         ----        ------
  --main                      0
  args [ o-]----->[]          1
     x [ 2 ]                  2
```

And finally, we print out the result of calling `inc` on `x` one last time. The call from `inc` returns with the following state:

```
    Stack         Heap        Output
    -----         ----        ------
  --main                      0
  args [ o-]----->[]          1
     x [ 2 ]                  2
  --inc                       3
     x [ 2 ]
     y [ 3 ]
```

and so we return 3 from `inc` and immediately print that result. Thus, the final state of the stack and heap as well as our output as we leave `main` is:

```
    Stack         Heap        Output
    -----         ----        ------
  --main                      0
  args [ o-]----->[]          1
     x [ 2 ]                  2
                              3
```

This seems like a lot of work to trace through such simple code, but all these details are necessary to ensure that we arrive at the correct result. This sort of attention to detail is critical for trying to reason about more complicated code where lots of variables and complex data structures are at play.

3.1.3 Objects and the Heap

Recall that values of strings and arrays are allocated on the heap rather than on the stack. Variables of these types are actually references to to these heap-allocated values. For example, after executing the following code snippet:

```
String s = "hello world!";
int[] arr = new int[] { 2, 4, 6 };
```

The state of the stack and heap are:

```
  Stack             Heap
  -----             ----
 s   [ ]-----> "Hello world!"
 arr [ ]-----> { 2, 4, 6 }
```

This logic applies to all objects. All objects are allocated on the heap and variables of these types are actually references to these values.

For example, consider a `Counter` class:

```
public class Counter {
    public int value;

    public Counter() {
        this.value = 0;
    }

    public void increment() {
        this.value += 1;
    }
}
```

Let's step through some code using `Counter` and explore how the stack and heap change over time. First, let's initialize a counter:

```
Counter c1 = new Counter();
```

We create a local variable for `c1` on the stack. Its eventual value is the result of the right-hand side of the assignment which is a `new` expression. When evaluating a 'new' expression, the following happens:

1. We evaluate the arguments of the `new` expression down to a value.
2. We allocate space on the heap for a `Counter`.
3. We then invoke the constructor of the `Counter` like a method or function.
4. We return a reference to the newly allocated and initialized `Counter` when exiting the constructor.

After the second step of this process, we enter the 'Counter' constructor with the following stack and heap:

```
  Stack                 Heap
  -----                 ----
 c1   [ ? ]  |-----> Counter
 -- Counter  |     value [ 0 ]
 this [   ]--|
```

We allocate a stack frame for the constructor call. The constructor has no arguments, but recall that we use `this` to refer to the newly created `Counter` within the constructor. In memory, we realize this by treating this as a variable—think of it as a hidden, first parameter to the constructor. This variable is loaded with a reference to the newly allocated Counter on the heap. Note that the heap allocation is a chunk of memory containing not only a field for the value field of the Counter but also a tag stating that the chunk of memory is indeed a Counter. Therefore, each object has a notion of its own type at runtime which we can query using the instanceof operator:

```
System.out.println(c1 instanceof Counter);    // prints "true"
```

After returning from the constructor, the stack and heap look like this:

```
  Stack                Heap
  -----                ----
 c1    [  ]---------> Counter
                     value [ 0 ]
```

Let's create another Counter:

```
Counter c2 = new Counter();
```

After executing the constructor, we arrive at the following stack and heap:

```
  Stack                Heap
  -----                ----
 c1    [  ]---------> Counter
                     value [ 0 ]
 c2    [  ]---------> Counter
                     value [ 0 ]
```

Note that c1 and c2 each have their own distinct value field, reinforcing the idea that each object has their own copy of the fields specified by their associated class.

Next, let's increment c1:

```
c1.increment();
```

Method invocations operate similarly to functions: we evaluate arguments, create a stack frame, and copy over arguments. The stack and heap look like this after entering the increment method:

```
  Stack                Heap
  -----                ----
 c1    [  ]---------> Counter <-----
                     value [ 0 ]    |
 c2    [  ]---------> Counter       |
                     value [ 0 ]    |
 -- increment                       |
 this [  ]------------------------|
```

While increment takes no parameters, the this variable is available in increment just like how it is available in the constructor. We load the this variable with (a reference to) the Counter object that we called increment on; in this case it is c1. Therefore, the effect of the single statement in Counter is to mutate c1's value field. After returning from increment, we arrive at this stack and heap state:

```
  Stack                 Heap
  -----                 ----
 c1    [  ]---------> Counter
                    value [ 1 ]
 c2    [  ]---------> Counter
                    value [ 0 ]
```

Note again that `value` fields of `c1` are distinct: we only incremented `c1` with that call to `increment` and did not modify ‘c2‘.

Exercise (Stack-and-Heap Tracing) Consider the following code that utilizes the `Counter` class:

```java
public class Program {

    public void doWork(Counter c) {
        c.increment();
        c = new Counter();
        // Point A
        c.increment();
        c.increment();
    }

    public static void main(String[] args) {
        Counter c = new Counter();
        doWork(c);
        // Point B
    }
}
```

1. Give a stack-and-heap diagram describing the state of memory at `/* Point A */` in the code above.
2. What is the value of `c`’s `value` field at `/* Point B */`?

3.2 Pre-Conditions, Post-Conditions, and Invariants

With appropriate mental models of computation, we can now reason about the behavior of our code. To do so, we specify *properties* about how our programs should behave and then *prove* that our programs obey those properties. Because our programs are complex, it is usually infeasible to directly prove their correctness. Instead, we usually define and prove several properties that, when taken together, imply the correctness of our program, or at the very least, give us high confidence that our program is performing as desired.

Reasoning about program correctness is particularly important for a pair of reasons

- **Ensuring quality code.** Bugs cost money, and bugs cost more money the later they are found in the development process. Finding a logic bug in our reasoning during program development is less costly than discovering it once the product has been deployed.

- **Driving program design**. It is instrumental in program design to have a crisp idea of how your program ought to behave. This knowledge helps concretely when actually designing and writing the program. As a simple example, suppose that you are writing a function that sums up the elements of the an array. You might have the following set up:

  ```
  int sum = 0;
  for (int i = 0; i < arr.length; i++) {
      // ??
  }
  ```

  If you recognize the *loop invariant* that sum always contains the sum of the numbers we've seen so far", then writing the loop in such a way to maintain the invariant is easy:

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

  Furthermore, you know your loop does the right thing because your loop invariant that you've ensured guarantees that sum contains the desired value.

3.2.1 Pre-Conditions and Post-Conditions

We can view the signature of a function or a method as a *contract* between the user of the function, the *caller*, and the and implementor of the function, the *callee*. In this view, a *pre-condition* of a function is an obligation that the caller must fulfill. In contrast, a *post-condition* of a function is an obligation that the callee must fulfill provided that the caller fulfills the pre-conditions of the function. For example, consider a function that computes $n!$ called `factorial`:

```
/**
 * @param n the input where n >= 0
 * @return n!
 */
public static long factorial(int n) { /* ... */ }
```

The pre-conditions to `factorial` are:

- The caller provides a single argument.
- That argument is an integer.
- That argument must be non-negative.

If the user meets these requirements, then the post-condition to `factorial` is the function produces $n!$ as expected. Note that some of the pre-conditions and post-conditions are enforced by Java's type system, *e.g.*, the function takes one argument that is an `int` and returns a `long`. Others are not enforced by the type system and are documented explicitly, *e.g.*, the argument must be non-negative and the function produces `n!`. Still others are not even documented as they are understood by convention. For example, because `factorial` returns a `long`, it will only work as long as $n \le 49$ (because $50! > 2^{63} - 1$, the maximum value a `long` can hold).

From the caller's perspective, pre-conditions and post-conditions inform the caller how to use a function and what to expect. From the callee's perspective, pre-conditions and post-conditions inform the callee what cases their function must be able to handle. In particular, by specifying a series of pre-conditions, the callee restricts these cases, making their job of implementation more managable. In the extreme case, pre-conditions keep the callee from having to "make up" outputs in the presence of invalid inputs such as negative values of *n* for `factorial`.

3.2.2 Small-Scale Reasoning

Pre-conditions and post-conditions allow us to reason about the correctness of *using* a method. However, how do we reason about correctness *within* a method? To do so, we must apply similar sorts of reasoning to program constructs other than method invocations.

Initial Values Let's start with the top of a method declaration, *e.g.*:

```
public void foo(int x, String s) {
    // ???
}
```

What can we say about the possible values of `x` and `s` as we enter the method? In the absence of any pre-conditions, the variables can take on any valid value that their types allow. This means that `x` can be any integer (recall that integers in Java are signed, 32-bit values) and `s` can be any string or `null` (as strings are a reference type). We frequently will want to avoid having to consider that an object may be `null` by assigning the function an appropriate pre-condition, for example:

```
/**
 * ...
 * @param s a non-null input string
 */
public void foo(int x, String s) {
    // ???
}
```

Assignments, Initialization, and Sequencing Consider the variable initialization/assignment statement:

```
int y = 5;
```

What do we learn about our program *after execution* of the initialization? Variable initialization introduces a new variable along with a value for that variable. So after the initialization, we know that `y` is `5`. However, what if we do the following assignment instead?

```
int y = x;
```

From above, we know that `x` is any integer. Therefore, after the assignment, `y` is equal to `x`, also any integer. Whatever value `y` had before is now gone; `y` now has the value `5`.

This seems like a trivial discovery, but this sort of reasoning gets more complicated as we chain together statements. For example, what about the following chain of assignments?

```
int y = 5;
x = x + y;
```

On the first line, we establish that y is 5. On the second line, we add y to x, so now what can we say about x? We can say several things:

- Coarsely speaking, x is still any integer value.
- More specifically, x is 5 greater than it originally was.
- In terms of ranges, x is any integer value in the range Integer.MIN_VALUE + 5 to Integer.MAX_VALUE assuming that we do not overflow x.

Note that to arrive at these conclusions, we *compose* our reasoning between the two statements. We analyze the first statement and then take what we learn from that statement (y is 5) and use it to analyze the second statement (x is incremented by 5). This composition of reasoning between statements is the cornerstone of reasoning about imperative programs.

Conditional Statements Consider a basic conditional statement:

```
if (x < 5) {
    // ??
} else {
    // ??
}
```

What can we say about x during execution of the conditional? If we happen to go into the if-branch, we know because of how the conditional operates, that the boolean expression must have evaluated to true. This means that inside the if-branch, x is certainly less than 5. Likewise, if we go into the else-branch, we know that the boolean expression must have evaluated to false. This means that inside the else-branch, x is certainly equal to or greater than5. Thus we can note that:

```
if (x < 5) {
    // Here, x < 5
    // ...
} else {
    // Here, x >= 5
    // ...
}
```

Regardless of the branch we go into, we know that control flows to the line after the if-statement. Therefore, the if-statement can introduce some uncertainty into what we know about our variables. Let's now make our conditional slightly more interesting.

```
if (x < 5) {
    y = 0;
} else {
    y = x + 1;
}
```

Now what can we say? If x < 5, then y is set to 0. Otherwise, if x >= 5, then oy is set to 1 greater than x. Combining both pieces of information, we can conclude that after the if-statement that y is either 0 or it is at least 6.

Generalizing, we note that x < 5 could be any boolean expression b. Therefore, we can reason about the conditional as follows:

```
if (b) {
    // we know b is true...
} else {
    // we know b is false...
}
// we know either the if-branch or the else-branch happened...
```

Importantly, after the conditional, we know that we entered one of the two branches. Therefore, after the conditional, we need to consider both situations in which we entered either the if-branch or the else-branch.

We can generalize this behavior to the alternative forms of conditional statements that Java offers. For example with multiple else-if branches, we can reason as follows:

```
if (b1) {
    // we know b1 is true...
} else if (b2) {
    // we know b1 is false, b2 is true...
} else if (b3) {
    // we know b1, b2 is false, b3 is true...
} else {
    // we know b1, b2, b3 is false...
}
```

Note how this contrasts with the absence of else-branches:

```
if (b1) {
    // we know b1 is true...
}
if (b2) {
    // we know b1 is true, nothing about b1...
}
if (b3) {
    // we know b3 is true, nothing about b1, b2...
}
```

As a concrete example, we'll commonly want to establish that a reference type is non-null before using it. The following snippet of code guarantees this:

```
if (s == null) {
    s = "default";
}
```

In the absence of a pre-condition or some other knowledge about s, it could have any string value, including null. If s is null, we enter the if-branch and assign it a non-null value. Otherwise, s was already non-null by definition and we skip the conditional altogether.

Loops How do we reason about loops? Unlike conditionals, they do not have ”straight-line” behavior—by definition, they loop! What can we say about a program after a loop terminates?

We establish such properties as *loop invariants*. Loop invariants are properties that hold:

- *Before* execution of the loop.
- *After each iteration* of the loop.
- *After* execution of the loop.

Note that the first two conditions implies the third. If a property holds before we enter a loop and then holds for each iteration of the loop, then it must hold after the loop finishes—the proof follows by *induction* on the number of iterations of the loop.

As an example, consider our motivating example from the beginning of this chapter. Suppose that we have the code:

```
int[] arr = /* ... */ ;
int sum = 0;
for (int i = 0; i < arr.length; i++) {
    sum += arr[i];
}
```

Let our loop invariant be that ”sum contains the sum of the numbers of arr that we have visited so far”. Our loop invariant holds initially—we haven’t seen any numbers yet so the sum is zero. On every iteration of the loop, we add the current number in the array to the sum. This maintains the invariant that the sum is indeed the sum of the numbers we have seen so far. Therefore, we know that the invariant is preserved and sum indeed contains the sum of the numbers found in the array.

Class Invariants Loop invariants are properties that are always true of a loop. We verify that they hold by ensuring that the property holds initially and then on every iteration of the loop. We can extend similar sorts of reasoning to verify the class invariants that we discussed when we talked about classes earlier. A class invariant is like a loop invariant except that instead of being true of a loop, it is true of any instance of a class.

As an example of a class invariant, suppose that we writing a weight-tracking application and want to represent a person in this application as a class, `Person`. We might begin the definition of the class as follows:

```
public class Person {
    private int weight;
    // ...
}
```

One immediate class invariant on `Person` is that `weight` should never go negative, that is `weight >= 0`.

Like loop invariants, we verify that the class invariant holds *initially* upon creation of the object. We then verify that every method *preserves* the class invariant thereby ensuring that the invariant always holds. Note that our particular invariant relies critically on `weight` being `private` so that arbitrary clients cannot simply mutate `weight` to be negative. For example, the constructor of the `Person` might look like:

```
public Person(int weight) {
    this.weight = weight;
}
```

The `weight` field is initialized by whomever calls the constructor. However, if they pass in a negative weight, this will violate our class invariant. To prevent this, we can guard against a negative weight and either (1) set the weight to `0` or (2) *throw an exception*.

```
public Person(int weight) {
    if (weight < 0) {
        throw new IllegalArgumentException();
    }
}
```

An exception signals an *exceptional case* in the execution of our code that we may (or may not) be able to recover from. In this case, the user of the `Person` class has violated our class invariant by giving an inappropriate weight. We signal an error by *throwing* an exception object. The syntax of a `throws` statement is:

```
throw <expr>;
```

Where `<expr>` is an expression that evaluates to an `Exception` object. There are a number of such objects available in the standard library to represent a wide variety of common scenarios. `IllegalArgumentException` represents the situation where a caller of a method or constructor has passed along illegal values. Note that here we are constructing a new `IllegalArgumentException` object with a parameterless constructor. The `IllegalArgumentException` class also specifies a one-argument constructor that is a string that documents more precisely the error that occurred, *e.g.*,

```
throw new IllegalArgumentException("Negative weight given");
```

With this guard, we ensure that if we successfully create a `Person` object, then we know the class invariant holds.

Now we just need to verify that each method of the `Person` class preserves this class invariant. For example, consider a method to update the person's weight:

```
public void updateWeight(int delta) {
    weight -= delta;
    if (weight < 0) { weight = 0; }
}
```

`updateWeight` simply subtracts its argument from `weight`. However, our class invariant becomes broken if `weight - delta` is negative. The method fixes this up by setting the `weight` to zero in this case. Note that a class invariant may be broken *in the middle of a method*. However, the class invariant must hold true by the time the method exits.

3.3 Invariants and Program Design

In the previous lab, we reasoned about programs using *pre-conditions*, *post-conditions*, and *invariants*. We did this in order to convince ourselves that our programs were correct; usually these conditions implied the overall correctness property of our program that we wanted to enforce. However, these reasoning tools are not just useful for ensuring that a created program is correct after-the-fact. We can use them, in particular, loop invariants to decompose problems and write solutions to them that are correct by construction.

3.3.1 Iterative Design with Loop Invariants

As discussed previously, loop invariants are properties that obey three rules:

1. A loop invariant is true *entering* the loop.
2. A loop invariant is true *at the end of every iteration* of the loop.
3. A loop invariant is true *after execution* of the loop.

Where the first two rules imply the third. If we identify that a solution requires a repetitive action to be performed, we can summarize the results of this repetitive action into a *loop invariant* that our loop will need to fulfill. By doing this, designing the loop becomes significantly easier.

As an example, consider the simple problem of writing a non-recursive version of the function `int factorial(int n)` that returns $n!$. For now we'll assume as a pre-condition that the user only passes `n >= 0` so that we do not need to worry about error-handling. This function clearly requires repetitive behavior, multiplying successive numbers from `1` to `n`, insinuating that a for-loop is a good construct to use.

```
int factorial(int n) {
    // ...
    for (int i = 1; i <= n; i++) {
        // ...
    }
    // ...
}
```

However, how do we fill in this skeleton? First we note that we need to perform some calculation on each number "visited" in the for-loop. This requires an additional local variable that we'll call `ret`. We initialize `ret` before the loop and return it afterwards.

```
int factorial(int n) {
    int ret = // ...
    for (int i = 1; i <= n; i++) {
        // ...
    }
    return ret;
}
```

Now, what loop invariant shall we enforce of our program? It needs to deal with the contents of `ret` and the behavior of the program we want to write. If our program needs to calculate `n!` and proceeds through the numbers `1` through `n` to do so, the following is a reasonable loop invariant to enforce:

> `ret` contains the product of the numbers from `1` to `k` where `k` is the count of numbers we have seen so far.

In light of this, we can apply our three rules of loop invariants to design our program:

1. **Entering the loop**. When we first enter the loop, we haven't seen any numbers yet. Therefore, we need to choose an initial value for `ret` that reflects this. We know that `0! = 1`, therefore we choose `ret` to be `0`.

2. **At the end of iteration of the loop**. On the `i`-th iteration of the loop, we see number `i`. How do we update `ret` so that our loop invariant holds? We merely multiply `ret` by `i` and store the result back in `ret`!

3. **After execution** By satisfying the two previous rules—preserving the invariant while entering the loop and on every iteration—we know that the loop invariant is preserved when the loop exits. Interpreting the loop invariant, we see that this implies that `ret` contains `n!`!

This *invariant-driven design* results in the following program:

```
int factorial(int n) {
    int ret = 1
    for (int i = 1; i <= n; i++) {
        ret *= i;
    }
    return ret;
}
```

Of course, you could have written this loop without explicitly going through process. But by following this invariant-driven design process mechanically, we can consistently produce loops that are correct, especially when the programs themselves get more complicated!

Exercise (Loop Invariant Verification) Consider the following implementation of the `max` function:

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

Give an appropriate loop invariant for `max` in terms of `ret` that implies the correctness of the `max` function if it holds. Does the `max` function enforce this loop invariant properly?

Chapter 4

Complexity

To analyze the correctness of a computer program, we reasoned about the flow of variable values throughout that program and verified logical properties using this information. In addition to correctness, another key property of computer programs we wish to understand is their *complexity*. We can break up complexity into two components:

1. *Temporal Complexity.* How long does a program take to execute?
2. *Spacial Complexity.* How much memory does a program take to execute?

To get a handle on how to analyze the complexity of our programs, we'll build up an appropriate mathematical model of our programs that accounts for complexity. First, we'll study how to *identify* and *count* the important operations that our programs perform in terms of their inputs. Then we'll learn how to *characterize* how these amounts change as the size of our inputs grow.

4.1 Real-world Measurement Versus Mathematical Models

To measure the amount of time that a program takes, we can simply use a clock and measure execution of a program from start to finish. In practice, this is only true way get a sense of what the *real* execution time of a program, independent of any mathematical models we develop about our program. Such a mathematical model is necessarily *incomplete*, it cannot capture all the effects that go into the performance of a program, so it is an *approximation* of the true run time of a program. However, even though this is the case, mathematical models are usually preferable to work with than with the physical process of time itself. This is due to a variety of reasons:

- Computers are fast enough that virtually all algorithms on small inputs perform identically.
- At the same time, developing large enough inputs that deviations appear in a program may be impractical for certain problems.
- To see trends in particular programs, we must run performance tests over many sorts of inputs which can be time-consuming, especially if the programs take a long time to run.
- The parts of a program that we want to analyze may not be amendable to real-world timing, *e.g.*, because that portion of the code is deeply intertwined in the system and cannot be isolated.
- Mathematical models can be platform-independent, or to put it another way, the actual run time of a particular program on a particular machine depends on that machine's configuration which may not be easily duplicable.

To this end, we develop and study mathematical models for understanding the complexity of our programs. Even though our focus is on these models rather than real-world measurement, it is important to keep in mind that such models are not the end-all-be-all of understanding program performance. These models are just one additional tool in your toolbox that you should be able to use when appropriate.

4.2 Identifying Relevant Operations

Each of the operations that our programs perform costs time which contribute to the overall run time of the program. Therefore, rather than measuring the actual time a program takes, we can count the number of operations the program performs. However, different operations take different amounts of time, *e.g.*, the cost of function call is (usually) more than accessing an element of an array. Rather than trying to get a more precise characterization of the run time of a program in terms of all the operations it performs, we can *approximate* its behavior by only counting the most "interesting" operations it performs.

For example, consider the following method:

```java
public static boolean contains(int[] arr, int k) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == k) {
            return true;
        }
    }
    return false;
}
```

This method performs a variety of potentially-interesting operations: variable initialization and assignments, comparisons of integers and returning values. In contrast, the variable initializations (the parameters `arr` and `n` and the local `i`) are uninteresting because:

1. They don't directly deal with what the function is computing (whether `n` is contained in `arr`).
2. The number of times they occur does not vary with the size of `arr` and `n`.

In contrast, the array accesses (`arr[i] == n`) are interesting precisely because they are the heart of the `contains` operation, and the number of such operations depends on the size of the input array. Consequently, reporting the number of variable initializations that `contains` performs is *less interesting* than the number of array accesses because it doesn't give us an accurate sense of how the method behaves. Thus, we would build our model of the method around the number of array accesses instead.

In summary, to build a model of the complexity of our program, we must identify which operations we wish to count as relevant. Such relevant operations (usually) directly perform the computation that is the intent of the program and they have dependence on the input in some way, *e.g.*, the size of some input integer or array.

When there are multiple relevant operations, our choice of which operations to count is arbitrary. Therefore, we choose the operation that makes our further calculations easier. For example, consider the following method:

```java
public static int addAdjacentPairs(int[] arr) {
    int sum = 0;
    for (int i = 0; i < arr.length - 1; i++) {
        sum = sum + arr[i] + arr[i + 1];
    }
}
```

Two choices of relevant operations to count are:

1. The number of assignments to `sum`.
2. The number of array accesses.

Clearly there is a relationship between the two operations—for every assignment to `sum` there are two array accesses. However, we ought to choose assignments for the simple reason that there are less of them to count.

4.3 Counting Operations

Once we have identified which operations we wish to count, then we need to go about the business of counting them. For example, consider the following method:

```
public static void swap(int[] arr, int i, int j) {
    int temp = arr[i];    // read
    arr[i] = arr[j]       // read, write
    arr[j] = temp;        // write
}
```

If we identify that we wish to count *array accesses*—both *reads* and *writes* to the input array—then this `swap` method performs 4 such operations, annotated in the comments above. Note that because our programs execute in a sequential manner, we simply count up the number of operations that occur in each statement. If we wanted to count the number of array operations of the following code snippet:

```
int[] arr = new int[10];
// ...
swap(arr, 0, 1);    // 4 ops
swap(arr, 10, 20);  // 4 ops
```

It would be the sum of the two `swap` calls—8 array operations overall.

Note that these counts so far have not depended on the size of the array. In contrast, consider the following method:

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

The number of array operations `sum` performs is the number of times the for-loop runs times the number of array operations performed in each iteration. One array access is performed during each iteration and we iterate through the loop `arr.length` number of times. Therefore, the `sum` method performs n operations where n is `arr.length`.

We express the number of operations formally as a *mathematical function* $T(n)$ where n is the size of the input that we have identified in our method.[1] This function serves as our *model* of our method's behavior. For example, we may describe the `sum` method as performing $T_1(n) = n$ operations where n is the size of the input array. Analogously, the function describing the run time of the `swap` function is $T_2(n) = 4$ where n is the size of the input array. Note that while `swap` has three parameters—`arr`, `i`, and `j`—we only identify the length of `arr` as the input to our model. Like choosing which operations to count, we must also ensure that we choose appropriate inputs to our mathematical function so that our model is accurate.

4.3.1 Counting Operations in Loops

Because a loop, by design, performs repetitive behavior, we can derive a formula for the number of operations any loop performs:

$$\text{Total Ops} = \text{No. of Iterations} \times \text{No. of Ops in Loop}$$

This is an instance of the *product rule* from a field of a mathematics called *combinatorics*, the study of counting. We saw how to apply this with the `sum` function:

$$\begin{aligned}\text{No. of Iterations} &= n\\ \text{No. of Ops in Loop} &= 1\\ \text{Total Ops} &= n \times 1 = n\end{aligned}$$

where n is the size of the input array. This formula generalizes to loops with irregular bounds and odd numbers of operations per loop, for example:

```
for (int i = arr.length - 1; i >= 2; i -= 3) {
    arr[i] = arr[i-1] - arr[i-2];
}
```

Here, the total number of iterations is difficult to see at first glance because the termination condition and decrement step are non-standard. If we write a few examples down:

- $n = 0$, no. of iterations = 0
- $n = 1$, no. of iterations = 0
- $n = 2$, no. of iterations = 1
- $n = 3$, no. of iterations = 1
- $n = 4$, no. of iterations = 1
- $n = 5$, no. of iterations = 2
- $n = 6$, no. of iterations = 2
- $n = 7$, no. of iterations = 2
- $n = 8$, no. of iterations = 3

[1]Note that the name of the function T and the input n are arbitrary. We could name them anything, *e.g.*, $f(x)$ is a more traditional notation for an arbitrary function. Throughout, we'll use the convention of T to represent a time complexity function, S to be a space complexity function, and the input of these functions to be n.

We can derive an explicit formula for the number of iterations: $\lfloor \frac{n+1}{3} \rfloor$ where $\lfloor - \rfloor$ is the *floor function* which rounds its argument down to the next whole number. Thus our formula for the total number of operations is $\lfloor \frac{n+1}{3} \rfloor \times 3$ because the loop performs three array accesses—two reads and one write—with each iteration.

The formula also works for nested loops where we simply calculate the total number of iterations of the inner-most loop and work outwards from there. For example, consider the following doubly-nested for-loop:

```
int sum = 0;
for (int i = 0; i < arr.length; i++) {
    for (int j = 0; j < arr.length; j++) {
        sum = sum + i + j;
    }
}
```

We know that the inner for-loop performs $3n$ additions (two in the update of sum and one to increment j) where n is the length of the input array. Thus, we know that the outer for-loop performs $(3n+1) \times n = 3n^2 + n$ such operations by accounting for the operations done by the inner loop and the increment of i.

The same technique applies when the bounds of the loops are different, for example, the slightly modified example:

```
int sum = 0;
for (int i = 0; i < arr.length; i++) {
    for (int j = 0; j < i; j++) {
        sum = sum + i + j;
    }
}
```

Now the inner loop's range depends on the current value of the outer loop variable. Again, going through examples to get a sense of the pattern of iteration:

- $i = 0$, No. Iterations of Inner Loop = 0
- $i = 1$, No. Iterations of Inner Loop = 1
- $i = 2$, No. Iterations of Inner Loop = 2
- $i = 3$, No. Iterations of Inner Loop = 3

Thus, if the length of the array is n, the inner loop will perform $1 + 2 + \cdots + n$ iterations. We can use summation notation to concisely write down this pattern:

$$1 + 2 + \cdots + n = \sum_{i=1}^{n} i$$

Euler gives us an explicit formula for this summation:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Thus, the total number of additions performed by this loop is

$$3 \cdot \frac{n(n+1)}{2} + n$$

where every iteration of the inner loop produces three additions and there are $\frac{n(n+1)}{2}$ such iterations. The additional n comes from the n increments of the outer loop variable.

Indeed, with arbitrary nestings of loops and bounds, we'll sometimes derive complicated nestings of summations. For example, we can express the number of array operations of the following triply-nested loop:

```
for (int i = 0; i < arr.length; i++) {
    for (int j = i / 2; j < arr.length - 1; j++) {
        for (int k = arr.length - 1; k > j; k--) {
            // 1 array operation performed
        }
    }
}
```

with a straightforward translation of loop bounds to summation bounds:

$$\sum_{i=0}^{n-1} \sum_{j=\frac{i}{2}}^{n-2} \sum_{k=n-1}^{k>j} 1$$

Note how the bounds of the for-loops translates directly into the bounds of the summations. The only complication to consider is that our for-loop bounds are typically exclusive on the upper bound whereas a summation is inclusive. In this particular case, we set up our summations to match the values that the iteration variables of the loops take, for example, 0 to $n - 1$ in the case of the outermost for-loop. In general, though, we may find it more convenient to choose different, yet equivalent range of numbers, for example, 1 to n in order to match a summation identity that we know of.

Frequently, this results in summations that are not trivial to simplify without summation identities and algebraic manipulation. Luckily, for the purposes of asymptotic analysis which we discuss shortly later, we don't need to be so precise with our counting because we are concerned more about the *trend* in the number of operations performed as the input size increases rather than the exact amount.

Eventually, we will reach the point where we can apply *informal reasoning* to our analysis and note that there are simply three nested for-loops with linear-like behavior and conclude that the number of operations behaves like $T(n) \approx n^3$. However, it is important to understand the fundamental mathematics involved in this analysis so that you can understand its corresponding strengths and weaknesses.

4.4 Cases of Execution

With this machinery in place, let's revisit our `contains` function:

```
public static boolean contains(int[] arr, int k) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == k) {
            return true;
        }
    }
    return false;
}
```

What is our model $T(n)$ of the time complexity of this function? Again, we choose to model the complexity by counting the number of array accesses of `arr` so n is the size of the input array. However, the analysis here is slightly trickier than `swap` and `sum`. This is because the number of array accesses depends not just on n but ultimately *where the contained element is in the array*, if it is in the array at all. How can we reconcile this problem?

To do this, we need to perform *case analysis* on the execution of the function. More specifically, we'll define three interesting cases—the *best case*, the *worst case*, and the *average case*—make assumptions about the input based on these cases, and then proceed to build our model.

The *best case* assumes that our input is set up in a way that leads to the fastest execution of the program possible. In the case of `contains`, this occurs whenever the requested element `k` is at the front of the array. In this situation, our best-case model for the function is $T(n) = 1$ because our for-loop only runs for one iteration.

The *worst case* assumes that our input is set up in the worst way possible, leading the slowest execution of the program. With `contains`, this occurs when the requested element `k` is either the last element in the array or is not in the array at all. In either case case, the for-loop runs n times (where n is the length of the input array), resulting in the model: $T(n) = n$.

The *average case* assumes something between the best and worst case. On average, what does the input look like and thus what sort of performance can we usually expect from the function? For `contains`, if our element has equal probability of being in any position in the array, its expected index is $\frac{n}{2}$. In this situation, the `for`-loop runs $\frac{n}{2}$ times resulting in the model: $T(n) = \frac{n}{2}$.

The case that we want to use depends on the circumstances and the questions that we are asking about our program. Furthermore, analyzing the cases may be trivial in some cases and non-trivial in others. For example, while the average case may be desirable in most circumstances, it may be difficult to assess the "average" case for a particular problem. In these situations, we may have to resort to using the best and worst cases to put a *lower* and *upper* bound on the performance of our program.

Exercise (Operation Count) Consider the following method:

```java
public static int max(int[] arr) {
    if (arr.length == 0) {
        throw new IllegalArgumentException();   // ignore this case
    } else {
        int ret = arr[0];
        for (int i = 1; i < arr.length; i++) {
            if (ret < arr[i]) {
                ret = arr[i];
            }
        }
        return ret;
    }
}
```

Build three mathematical model of the `max` method describing its worst, best, and average case performance. Identify the relevant inputs to the method, operations to count, and give functions that describes the number of operations the method performs in terms of the inputs that you identify.

4.5 Big-O Notation

So far we built up intuition about what it means to perform complexity analysis and learned how to model the number relevant operations a program performs using mathematical functions. However, as mentioned previously, computers are fast enough that on small inputs, virtually any set of algorithms that solve the same problem will perform identically. Rather than worry about small inputs, we would like to understand how our program behavior *scales* when it is given larger inputs. The study of the *limiting* behavior of our programs is called the *asymptotic analysis of algorithms* and the main tool we use to express this behavior is *Big-O notation.* In this section we'll study this mathematical formalism and tie it back to the informal reasoning we will need to perform on a day-to-day basis.

4.5.1 Growth of Functions

We model the complexity of computer programs using mathematical functions as we saw in the reading and lab yesterday. We can categorize the different mathematical functions in terms of how they grow as their inputs grow.

- **Constant Functions** are those functions that do not depend on their input. For example $f(x) = 300$ is a constant function that evaluates to 300, irrespective of its input. Our `swap` function from before performed a constant number of array operations.
- **Linear Functions** take the form $f(x) = mx + b$ where m and b are constants. They correspond to lines. For example, walking an array takes linear time.
- **Quadratic Functions** take the form $f(x) = ax^2 + bx + c$ where a, b, and c are constants. They correspond to curves. Functions with quadratic complexity arise, for example, when we must perform an operation involving all possible pairs of a collection of objects. If there are n objects, then there are $n \times n = n^2$ operations that must be performed.
- **Cubic Functions** take the form $f(x) = ax^3 + bx^2 + cx + d$ where a, b, c, and d. They correspond to curves with an inflection point and have a slope greater than a quadratic function. Functions with cubic complexity arise, for example, when we must perform an operation involving all possible triples of a collection of objects. Like the quadratic case, if there are n objects, then there are $n \times n \times n = n^3$ operations to be performed.
- **Polynomial Functions** generalizes all of the previous functions discussed so far. A polynomial has the form $f(x) = \sum_i a_i x^i + c$ where each a_i and c are constants. We'll usually lump quadratic and cubic functions under the "polynomial" functions and be more specific when we want to talk about linear and constant functions.
- **Exponential Functions** take the form $f(x) = ab^x$ where a and b are constants. They also correspond to curves but with a steeper slope. Exponential functions arise, for example, when we have to consider all possible subsets of a collection of objects. For a collection of n objects, there are 2^n possible such subsets.
- **Factorial**, $f(x) = x!$, corresponds to the number of possible orderings or *permutations* of x elements. If our program needs to generate or consider all of the permutations of a collection of n elements, then its runtime will be $n!$.
- **Logarithmic Functions** take the form $f(x) = \log x$. When using log we usually assume the base of the logarithm is 10 (so that $\log 10^3 = 3$). However, in computer science, we usually assume log is

base 2. It will turn out the base of the logarithm is usually irrelevant for our purposes of asymptotic analysis because via the change-of-base rule—$\log_a x = \frac{\log_b x}{\log_a b}$—logarithms of different bases only differ by a constant amount (the term $\log_a b$ in the rule). Logarithmic functions arise when we are able to divide a problem into sub-problems whose size is reduced by some factor, *e.g.*, by half. When these problems are smaller versions the original problem, we call them "divide-and-conquer" problems and frequently use recursive design to solve them.

- **Linearithmic Functions** are "linear-like" functions by some logarithmic factor, *i.e.*, have the form $f(x) = x \log x$. Linearithmic functions arise when a divide-and-conquer sub-problems requires a linear amount of work. For example, the most efficient general-purpose sorting algorithms have this runtime.

4.5.2 Big-O Notation

When we perform complexity analysis, we would like to classify the growth behavior of our program according to one of the classes of functions listed above. We use *Big-O* notation to capture this fact. When we write $\mathcal{O}(f)$ for some function *f*, we refer to the set of functions that are all in the same growth class as *f*. For example, $\mathcal{O}(f)$ where $f(n) = n$, refers to the class of linear functions such as:

$$f_1(n) = 3n f_2(n) = 5n - 3 f_3(n) = 1 - n$$

If we therefore think of $\mathcal{O}(f)$ as a set of functions, we can write $g \in \mathcal{O}(f)$ to mean that function *g* belongs to the same class of functions that *f* belongs to (*i.e.*, the class denoted by $\mathcal{O}(f)$. The functions f, f_1, f_2, f_3 above are all in the same complexity class so $f \in \mathcal{O}(f_1)$, $f_1 \in \mathcal{O}(f)$, $f \in \mathcal{O}(f_2)$, $f_2 \in \mathcal{O}(f_1)$, *etc.*.

We can categorizing the complexity of our functions by using Big-O notation in tandem with the mathematical models we build to count the functions' relevant operations. For example:

- The number of array operations performed by the `swap` method is $T(n) = 4$ where n is the size of the input array. We can say that $T \in \mathcal{O}(1)$, declaring that the runtime of `swap` is in the constant complexity class.
- The number of array operations performed by the `sum` method is $T(n) = n$ where n is the size of the input array. We can say that $T \in \mathcal{O}(n)$, declaring that the runtime of `sum` is in the linear complexity class.

Note that when describing the complexity class, we use the simplest function in that class, *e.g.*, n instead of $2n$ or $3n + 5$. While using more complicated classes is technically accurate, such usage loses sight of the fact that we are referring to the class of functions that are linear rather than a particular such function. In particular, with the constant complexity class we write $\mathcal{O}(1)$ since it relates together all constant functions together.

4.5.3 The Formal Definition of Big-O

So far, we've developed an informal idea of Big-O—a classification of the growth rate of mathematical functions. Now let's unveil the specifics. The formal definition of Big-O is:

$$f \in \mathcal{O}(g) \iff \exists c > 0, n_0. \forall n \leq n_0. |f(n)| \leq c|g(n)|$$

What does this mean? First of all, for some function of interest *f*, we say that $f \in \mathcal{O}(()g)$, pronounced "*f* is oh-of-*g*" or "*f* is order *g*". This is true whenever there exists (∃) two constants c and n_0 such that for all (∀) n

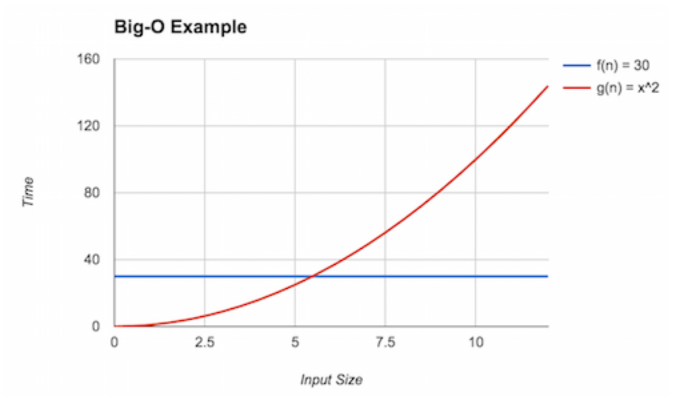
where $n \leq n_0$ the following inequality holds: $|f(n) \leq c|g(n)|$. That is $g(n)$ dominates $f(n)$ by some constant factor c after some starting input n_0.

$f \in \mathcal{O}(()g)$ captures the idea that f is *bounded above* by g.[2] To prove this fact, we must provide the two integers demanded by the existential:

- c, a constant factor that g is multiplied by and
- n_0, the minimum input size to consider.

such that for all input sizes greater than or equal to n_0, $f(n)$ is less than or equal to $c \cdot g(n)$. That is, from n_0 on, f is also smaller (or equal) to g within a constant.

For example, let's show that $f \in \mathcal{O}(g)$ where $f(n) = 30$ and $g(n) = x^2$. First let's examine a graph of the two functions: We can see that eventually g (the red line) dominates f (the blue line), but what is that



point? This is the point where $g(n) = x^2 = 30$. Solving for n yields $n = 30^{1/2} \approx 5.48$. Thus, we can claim that $n_0 = 6$ (rounding up to be safe) and $c = 1$. Here, we see the inequality holds because $f(6) = 30 \leq g(6) = 36$. With this, we can conclude that $f \in \mathcal{O}(g)$.

Note that Big-O provides an *upper bound* on the asymptotic complexity of a function. For example, in the above example $f \in \mathcal{O}(n!)$ as well. To see this, note that for $n \geq 5$, $f(n) \leq g(n)$. However, this is a *weak* upper bound because many other classes of functions are "smaller" than factorial, for example, polynomials and linear functions.

We always want to provide the *tightest* bound possible. However, because we are not analyzing pure mathematical functions but computer programs with arbitrary behavior, we will sometimes be unable to give a tight bound because it will be impractical, infeasible, or impossible to give a precise model. We will therefore resort to a less tight bound in these situations. For example, you may suspect that the program has quadratic complexity but have difficulty proving it, so instead, you may claim a cubic bound instead which may be easier to show.

4.5.4 Additional Asymptotic Bounds

Big-O notation gives us a way of describing the behavior of a function as its input grows. In mathematics, we formally define this as taking the limit of the function (typically to infinity). Thus, our techniques above are an example of *asymptotic analysis* where we analyze the limiting behavior of mathematical functions.

[2]The traditional way of writing $f \in \mathcal{O}(()g)$ is $f(x) = \mathcal{O}(g(x))$. However, equality notation here obscures the idea that $\mathcal{O}(g(x))$ denotes a *set* of functions—all functions that differ by $g(x)$ by at most a constant.

The definition of Big-O described previously has many parts to it. By changing this definition we obtain other useful notations for describing the asymptotic behavior of functions. For example, consider flipping the inequality relating f and g. We obtain the following definition:

$$f \in \Omega(g) \iff \exists c > 0, n_0.\, \forall n \leq n_0.\, |f(n)| \geq c|g(n)|$$

When it is f that dominates g, we write $f \in \Omega(g)$ (pronounced "f is big-omega of g"). Intuitively, this definition states that g is a *lower bound* of f. That is, f performs no better than g. For example, it is known that any comparison-based sorting algorithm is $\Omega(n \log n)$, *i.e.*, no such algorithm can perform better than $\mathcal{O}(n \log n)$.

If g serves both as an upper- and lower-bound for f, we write $f \in \Theta(g)$ (pronounced "f is big-theta of g").

$$f \in \Theta(g) \iff \exists c_1, c_2 > 0, n_0.\, \forall n \leq n_0.\, c_1|g(n) \leq |f(n)| \leq c_2|g(n)|$$

Here, g describes *exactly* the family of functions in which f resides. Often we would like to characterize our models with big-theta bounds. However, it is frequently onerous, difficult, or both to prove that g is a lower bound, so we frequently stick with big-O bounds instead of big-theta.

Finally, note in the statements above that we posit that *there exists* a constant that inflates g appropriately. This implies that there may exist some constants where the desired inequality—less than for big-O and greater than for big-omega—does not hold. We can make a stronger claim by asserting that the inequality holds *for all* constants rather than at least one:

$$f \in o(g) \iff \forall c > 0.\, \exists n_0.\, \forall n \leq n_0.\, |f(n)| \leq c|g(n)|$$
$$f \in \omega(g) \iff \forall c > 0.\, \exists n_0.\, \forall n \leq n_0.\, |f(n)| \geq c|g(n)|$$

These "little" versions of big-O and big-omega ("little-o" and "little-omega", respectively) make a strictly stronger claim than their big counterparts by asserting that the desired inequality holds, irrespective of the chosen constant. In practice, this means that f and g are in strictly different families of functions.

Exercise (Big-O) For each of the three mathematical models of the `max` method that you gave in the previous reading exercises, give a big-O bound on the asymptotic complexity of those models. Give the tightest bounds possible for each.

4.6 Recurrence Relations

So far, we have analyzed the complexity of programs that contain loops by straightforward counting, *e.g.*, if a loop executes n times and performs k operations on each iteration, the total number of operations performed is $k \times k$. However, what about recursive programs? How do we account for recursive calls within these programs? To build mathematical models for these programs, we introduce *recurrence relations*, a particular sort of mathematical function defined in terms of itself, just like recursive programs!

4.6.1 An Example: Factorial

Consider the standard recursive definition of factorial:

```
public static long factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return factorial(n - 1) * n;
    }
}
```

How do we analyze the complexity of this function? First we must choose what operations we will track and the corresponding ”input” to our model. We compute factorial by ”stripping” off one unit from the input, n, performing a multiplication in the process. Therefore, we should analyze this function by counting multiplications; our input should be n itself.

Note that our recursive definition of `factorial` is defined by a conditional on n and in general, you should recognize this as an instance of the general recursive program *skeleton*:

```
if (<base case condition) {
    // <base case>
} else {
    // <recursive case>
}
```

Our model will have similar structure. We will define it in terms of the number of operations performed at the base case (when $n = 0$) and at the recursive case (when $n \neq 0$):

$$
\begin{aligned}
T(0) &= \ldots \\
T(n) &= \ldots
\end{aligned}
$$

And now we need to give the number of operations that occur in each of these cases. $T(0)$ is straightforward: in the base case, we perform no multiplications, so $T(0) = 0$. However, what about $T(n)$? We perform one multiplication and then perform a recursive call where the input is reduced by one. Our definition for $T(n)$ reflects this exactly:

$$T(n) = 1 + T(n - 1)$$

Thus, our complete *recurrence relation* that models `factorial` is

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

4.6.2 From Recurrences to Big-O

So far, we have used Big-O notation to “summarize” the behavior of our model as its input grows in size. We cannot immediately use Big-O notation to summarize our recurrence relations because they are not in the form of one of our standard function classes. We therefore need to either (a) *solve* our recurrence relation and derive an explicit formula or (b) derive an *approximate* formula for our relation. Some relations are solvable (*i.e.*, have a closed-form solution), but many others are not, and thus we have to resort to approximations.

It turns out that our recurrence above has an easy closed-form solution that we can derive using the *substitutive method* in the follow manner:

1. First, let's expand the recurrence a few steps to get a feel for the pattern of its behavior:

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

2. Next, let's try to rewrite $T(n)$ in terms of the number of expansions or *unfoldings* that we perform of the function. Call this number k.

$$T(n) = 1 + T(n-1) = k + T(n-k)$$

   We arrive at this formula by noting that after k unfoldings, we add k 1s and make a recursive call to $T(n-k)$ in the final unfolding.

3. Finally, we note that the base case of our recursion is when $n = 0$. So we consider the case where the recursive call in the formula above results in the base case. This is when $n = k$ so $T(n-k) = T(n-n) = T(0)$. Plugging in for k gives us an explicit formula for the overall recurrence:

$$T(n) = k + T(n-k) = n + T(0) = n + 0 = n.$$

Thus, the explicit formula for the number of multiplications that `factorial` performs in terms of its input n is $T(n) = n$. In terms of big-O notation, we say that $T \in \mathcal{O}(n)$—that is, `factorial` takes *linear* time with respect to its input.

Exercise (Recurrence Relations) Solve the following recurrence, slightly modified from the reading:

$$T'(0) = 0$$
$$T'(n) = 1 + T(n-2)$$

and give big-O bound on the behavior of T'. In your analysis, assume that n is even and $n \geq 2$. Follow the outline of the proof in the reading as a guide. Does your big-O bound change if, instead of subtracting 1 or 2, you subtract any constant k (assuming $n \geq k$ and k divides n evenly)?

Part III

Structures

Chapter 5

Algorithmic Design

(This chapter is currently under heavy revision!)

All of our programs are composed of two parts:

1. Data.
2. Operations over that data.

We now turn our attention towards the these data structures and algorithms that make up our programs. Previously all

Of course, all but the simplest of toy programs contain many pieces of related data that we most organize in some way. We do so through various *containers* that efficiently hold data in various ways

Before we dive into the specifics of data structures and algorithms, however, we'll first take a step back and introduce some high-level concepts that will help structure our discussion for the rest of the course.

5.1 Data Organization

5.2 Algorithmic Design

Algorithm design is a daunting affair. You probably have experienced this first-hand on exams where you were asked to write a program to accomplish a task, *e.g.*, sorting the elements of an array. How do you go from a problem statement to a working algorithm? Much of the time, it feels like a game of chance; either you either "get it" or you don't with little room in between. And to be honest, this mindset has some merit. Algorithm design is a difficult task with a steep slope. In a heartbeat, you'll go from not knowing at all how to solve a problem to kicking yourself for not seeing the solution earlier. This is because most algorithms only require a single burst of inspiration, a spark of an insight, a key idea, *etc.*.

However, algorithm design is not entirely luck-based! There are ways we can be systematic about our design process so that we quickly and consistently hone in on a final solution. Here I want to touch on two complementary skills you will use in designing algorithms:

a Consulting Your Toolbox.

b Exploring the Problem Space.

5.3 Consulting Your Toolbox

One way of solving an algorithm is to know it already. This line of thinking is what motivates many books for "Cracking the Coding Questions" on internship or job interviews. If we know everything, then we'll never be stumped by an algorithms question!

Of course, there is clearly a problem with this line of reasoning. We can't possibly know the solution to every problem set before us beforehand. Rather than memorizing solutions to particular problems, we need to be able to generalize the solutions to problems that we have previously encountered so that we can apply those techniques to as many new problems as possible.

You can think of this process as building up an *algorithmic toolbox*, a collection of strategies and techniques that apply to large classes of problems. Because our computer programs consist of data and functions that manipulate that data, we can break up our toolbox into two parts:

a *Abstract data types* that organize our data so that we can efficiently access them as our program demands it, and *data structures* that provide concrete implementations of those abstract data types.

b *Programming patterns* that allow us to manipulate our data efficiently and in a readable manner.

5.3.1 Abstract Data Types

One dimension of your toolbox concerns how you organize your data. These various sorts of fundamental abstract data types and data structures are the focus of this course. An abstract data type defines (ADT) an *interface*, the allowable set of operations on data of a particular type. In contrast, a data structure provides a particular implementation for that abstract data type.

We organize our abstract data types broadly into four categories based on the relationships we maintain between the data:

- *Sequential types*, a linear ordering between data.
- *Hierarchical structures*, a parent-children relationship between data.
- *Mapping structures*, key-value relationship between data.
- *Graph structures*, arbitrary relationships between data.

For example, within the sequential types category, there are a number of abstract data types, including:

- Lists—Variable-size, homogeneous, sequential containers.
- Arrays—Fixed-size homogeneous, sequential, random-access containers.
- Tuples—Fixed-size heterogeneous, random-access containers.
- Queues—Homogeneous containers with first-in-last-out (FILO) behavior.
- Stacks—Homogeneous containers with first-in-first-out (FIFO) behavior.

Finally, each abstract data type has one or more data structures that implement that data type. In some cases, the implementing data structure is the same as the abstract data type. For example, an array defines an abstract data type but is also its own implementation. (In some contexts, you will see the term "Vector" used as the abstract data type that an array implements.) However, in most cases, there will be multiple data structures that implement a particular abstract data type. For example, the List abstract data type can be implemented by using a linked-based or array-based data structure.

When solving a problem, we must identify how we should organize our data. This in turn helps us identify which abstract data type we should use. Frequently, multiple abstract data types fulfill our needs, in which case, we may have to weigh trade-offs between them to choose one. Likewise, once we have identified an abstract data type, we will need to choose a particular implementing data structure, each with their own trade-offs, *e.g.*, performance between various operations or implementation complexity.

5.3.2 Programming Patterns

Once we've chosen how we represent our data, we must actually manipulate that data to obtain the desired result. We rarely invent all-new ways of manipulating data on the fly. Rather, we rely on number of *programming patterns* as a basis for our solution, and then refine it further to solve the particular problem at hand. These programming patterns are designed to accomplish some small task en route towards solving the overall problem. Examples of such patterns you have seen before:

- *Iteration*: using a for-loop to traverse an array.
- *Swapping*: swapping the contents of two variables.
- *Finger-pointing*: generalizing iteration across multiple containers by tracking indices into each container.
- *Link-walking*: maintaining a "current" pointer to traverse a linked structure.
- *Divide-and-conquerer*: decompose a problem into smaller versions of itself plus some extra work to fix everything up.

When solving a problem, you should first try to decompose your problem down into these more elementary patterns. That way you can get the "easy" parts of the problem out of the way to focus on the problem-specific stuff. Likewise, as you learn about new problems, you should be taking note of these patterns that arise so that you can reproduce them for future problems as necessary.

5.4 Exploring the Problem Space

Consulting your programming toolbox is all about organizing and accessing your prior programming knowledge to help you solve the problem at hand. However, because each problem is unique, you will inevitable need to come up with new knowledge. How can you do this in a systematic way?

My suggestion is to build a *habit of exploration* that you go to when you encounter a problem. This process has four steps:

1. Understand the problem.
2. Formalize the interface.
3. Come up with examples.
4. Solve the problem (and recur).

The first step, understanding the problems, involves identifying the high-level pieces that will go into our solution. What data must we process? Broadly, what approach will we take towards solving the problem? As you get better at designing algorithms, this first step becomes quite natural and automatic. However, as you begin your journey into algorithmic design, you should be very explicit about identifying at a high-level the data and techniques you will use to solve the problem in question.

Formalizing the interface to the problem, consists of taking the high-level concepts we identified in understanding the problem and beginning to enshrine them in code. For example, at this step, we choose the types that our data take on either primitives or new class types as necessary. Broadly, all problems that we solve will take in some input and produce some output. If we are, *e.g.*, writing a function to solve the problem, then we would codify these inputs as parameters and the output as a return value. Relevant, pre-conditions and post-conditions should also be documented in comments.

Coming up with examples of how your solution should behave on different inputs serves to purposes:

1. They help us verify that we are implementing the correct solution and,
2. They help us find patterns and insights into the problem we are trying to solve.

It is very easy to skip this, but for hard problems, I can't stress the importance of using examples to explore the problem. One of the best pieces of advice I ever received on algorithm design was the following:

> You cannot tell a computer how to a solve problem until, at a bare minimum, you know how to solve the problem yourself.

Writing input-output examples is how we learn how to solve the problem ourselves. When you do this, you should be exploring the range of inputs to the problem, in particular identifying a few common cases as well as all the relevant corner cases. Finding the corner cases is important because they help us identify base cases in our programs and test our knowledge of the problem space, *e.g.*, what happens if this function is fed an empty array or string? Identifying several common cases is also important because they help us see patterns in the code which then lead to generalizations of our solution strategy to a working algorithm.

(As an important aside, I'm drawing a subtle, yet important distinction between coming up with examples versus *tests*. A test enshrines an example into an artifact you can use to automate the process of verifying your program. If possible, you should go through process of realizing your examples as tests, *e.g.*, written in JUnit. However, you should not avoid this step just because a suitable testing environment is not available for your particular language!)

After you have gotten comfortable with the problem—you know how to solve it for yourself—you can now begin trying to understand how you might write a program to do that. To do this, you need to think of solving the program in terms of the basic programming patterns that we talked about earlier. For example, you might identify that to sort an array, you follow a "deck of cards" approach where you repeatedly find the smallest card and put it at the front of the deck. To translate this high-level approach into a computer program, we must decide what programming patterns best capture this behavior:

- Because we are performing a repetitive action over an array, *iteration* is important to traverse the array.
- *Finding the minimum element* of a subarray also seems important here.
- Finally, we likely need to *swap* elements of our array to emulate putting the minimal element at the front of the deck.

Now, our task of writing a sorting function is much easier; we now need to arrange these more elementary operations in a way that achieves our sorting behavior.

Naturally, as you get better at algorithmic design, you'll coalesce or skip some of these skips entirely. But you should do your best to follow these steps as you begin your journey into data structures and algorithms, that way you increase your consistency and get the most out of all the problems we'll explore in the reminder of the course!

Chapter 6

Sequential Structures

The first of the fundamental data types that we'll study are *sequential types*. A sequential type is simply a container for data where the container maintains an ordering between elements. As a result, a common operation we can perform over a sequential types is to *iterate* over the elements of the container in-order.

The canonical example of a sequential type built-in to most languages is the array. Arrays have the following properties in addition to being sequential:

- *Homogeneous*. An array holds elements of the same type.
- *Fixed-size*. Once created, an array's size cannot change.

Other sequential types differ along these two dimensions. For example, a heterogeneous fixed-size sequential type is called a *tuple*. We frequently utilize tuples of two elements, *pairs*, and tuples of three elements, *triples*. Tuples are built-in to some languages, but not Java. So we have to define a class for each tuple type that we're interested in using, *e.g.*, a `Point` class:

```java
public class Point {
    private int fst;
    private int snd;
    public Point(int fst, int snd) {
        this.fst = fst;
        this.snd = snd;
    }
    public int getFst() { return fst; }
    public int getSnd() { return snd; }
}
```

6.1 The List

Frequently, we do not know the number of elements we need to store into a sequence up front, for example, because the input comes from the user during execution of the program. Fixed-sized types are insufficient in these situations. We instead need a *variable-sized* type that can grow at runtime to accommodate our storage needs.

6.2 Linked- and Array-based Lists

6.3 Amortized Analysis

Recall the basic set up for an array list and its `add` method:

```
public class ArrayList<T> {
    private Object[] arr;
    private int front;
    public ArrayList() {
        arr = new Object[16];
        front = 0;
    }
    private void ensureCapacity() {
        if (front >= arr.length) {
            arr = Arrays.copyOf(arr, arr.length * 2);
        }
    }
    public add(T v) {
        ensureCapacity();
        arr[front++] = v;
    }
}
```

Analyzing the time complexity of `add` is tricky because its behavior does not follow the standard patterns we've seen up to this point. In particular, there are two cases that dictate the perform of `add`:

1. If `front < arr.length`, then there is room in the backing array for the new element. Adding the element onto the end of the list takes $\mathcal{O}(1)$ time since it consists of an array copy and an increment.
2. If `front >= arr.length`, then we must make room for the new element in the backing array. We do so by creating a new array that is twice the size, copy the elements over, and then copy in the new element. This takes $\mathcal{O}(n)$ time (where n is the number of elements in the array).

For an individual `add` operation, the best case is the former case and the worst case is the latter case. However, what is the average case? We usually define the average case in terms of the shape of the input, *e.g.*, assuming that the element that `indexOf` searches for lies in the middle of the list. Here, there is no "average" shape here—either the backing array is full or it isn't. This makes characterizing average difficult.

Certainly, calling the average case $\mathcal{O}(n)$ in line with the worst case is not incorrect (we are giving upper-bounds, after all), but such a bound isn't consistent with our intuition as to how `add` works. By doubling the array size on every growth, we hope that we will perform many more $\mathcal{O}(1)$ additions than $\mathcal{O}(n)$ additions, effectively making the runtime $\mathcal{O}(1)$. Our current definition of time complexity do not support this idea of analyzing the runtime of a method over many invocations—it only considers a single operation in isolation. Therefore, we need additional machinery to analyze `add` faithful.

This machinery is called *amortized analysis* where we look at not just a single call to `add`, but many calls to `add` to underhand its "average" complexity. There are three different basic techniques for performing amortized analysis:

- *The aggregate method* which directly characterizes the total complexity of a series of operations and then declares the average complexity to be the average of this amount.
- *The accounting method* which charges cheap operations additional time in order to pay for the time debt incurred by more expensive operations.
- *The potential method* which builds a potential function that describes the "time" built-up by cheap operations that can be used by more expensive operations. Unlike the accounting method, the potential function is only based on the current state of the data structure, not the history of operations that led to that point.

Here, we will use the accounting method to analyze `add` because it most closely works with our intuition that the cheap `add` operations should "pay" for the expensive `add` operations.

Consider an array list with starting capacity two and a series of add operations on it along with the number of array accesses they perform:

Add Op	Actual Cost
1	1
2	1
3	3
4	1
5	5
6	1
7	1
8	1

The third and fifth additions induce array doubling (corresponding to when the backing array is at two and four elements, respectively). Each of these perform $k + 1$ array accesses where k is the number of elements at the time of the copy plus one additional copy for the new element. The remaining additions perform a single array access.

The goal of the banker's method is to assign a single cost to all of the `add` operations c, the *charge cost*, such that the charge cost allows us to pay for the actual cost of successive add operations without going negative. For example, suppose we consider $c = 1$ which corresponds to charging each `add` operation one array access.

Add Op	Actual Cost	Charge Cost	Balance
1	1	1	0
2	1	1	0
3	3	1	**-2**

The balance column is the amount of cost left over after using the balance and charge cost to pay for the actual cost. Because the charge cost is one, the first two operations are paid for without leaving any balance left over. Unfortunately, this means that when we try to pay for the third operation, our balance is empty, so our charge cost of 1 is not enough to pay the actual cost (3) of the operation. This means that $c = 1$ does not work as a charge cost for `add`.

Next, let's try $c = 2$:

Add Op	Actual Cost	Charge Cost	Balance
1	1	2	1
2	1	2	2
3	3	2	1
4	1	2	2
5	5	2	**-1**

Note that by charging 2 array accesses per `add`, we build up enough of a balance to pay off the first expensive addition. However, the charge cost is not sufficient to pay for the second expensive addition.

Finally, let's try $c = 3$:

Add Op	Actual Cost	Charge Cost	Balance
1	1	3	2
2	1	3	4
3	3	3	4
4	1	3	6
5	5	3	4
6	1	3	6
7	1	3	8
8	1	3	10
9	9	3	4

With $c = 3$, we have enough for the second array copy (at the fifth operation) However, this charge cost also gives us enough left-over balance to pay for the third array copy (at the ninth operation), too! Furthermore, if we look at the left-over balance after each array copy operation, we see that we are left with 4, so it seems like this pattern might continue indefinitely. How can we prove this is the case?

Consider the cost of `add` when we copy an array. If the array contains `n` elements, we must pay one array access to copy each old element along with one addition array access to copy the new element. From this, we can justify our choice of $c = 3$ for each element added to the array

- 1 array access goes towards the actual insertion of an element.
- 1 array access goes towards the first time this element is copied into a larger array.
- 1 array access goes towards the insertion of one of the elements in the lower half of the array.

Note that if the currently copied array is size n, then $n/2$ elements have been added to the array since the last copy. The final charge accounts for the copying of the older $n/2$ into the new array.

With $c = 3$, we see that we can assign `n` a constant amount of time that is accurate over many operations of `add`. Therefore, we conclude that `add` has *amortized* $\mathcal{O}(1)$ complexity.

Chapter 7

Polymorphism

Previously, we developed two data structures that implemented the *list* abstract data type: linked lists and array lists. However, these implementations were unsatisfying along two dimensions:

1. Even though a linked list and array list provide the same functions to a user, they are not *interchangable* in user code. If the user wanted the flexibility of choosing between a linked list or array list (because they have different performance characteristics), they would need to duplicate code who's only difference would be the type of the list it uses.
2. The *carrier type* of either list implementation is fixed. If we wanted a list that held integers as well as a list that contained strings, we would need two separate implementations of the list that only differed in the type of the value at each node.

To solve both these problems, Java provides us mechanisms to write *polymorphic* code. Polymorphic code is code that can operate over multiple types. In particular, the two problems described above are addressed with two types of polymorphisms: subtype polymorphism and parametric polymorphism.

7.1 Subtype Polymorphism

The list abstract data type defined a number of operations that all "list-like" objects ought to implement:

- `int size()`,
- `void add(int x)`,
- `void insert(int x, int index)`,
- `void clear()`,
- `int get(int index)`, and
- `int remove(int index)`.

Our linked list and array list classes implemented these methods. However, there was no enforcement by the compiler that these classes actually implemented these operations. Furthermore, even though the two list implementations provided the exact same set of methods to the user, we could not interchange one list for another because they are different types. For example, if we have a method:

```
public int sumAll(LinkedList l) {
    int sum = 0;
    for (int i = 0; i < l.size(); i++) {
        sum += l.get(i);
    }
    return sum;
}
```

A programmer cannot pass an array list to this method even though it would work perfectly fine with one.

To this end, we need a programming mechanism for capturing the shared behavior between linked list and array list, enforcing that they indeed share this common behavior as well as allowing us to interchange them in our code. This mechanism in Java is called the *interface*.

An interface looks like a class declaration except that:

1. It uses the `interface` keywords in its definition rather than `class`.
2. It only contains (`public`) methods.
3. It does not provide any implementation of these methods; in other words, the interface only contains *method signatures*.

For example, here is the realization of our list ADT as a Java interface called `List`. Note that like classes, a `public` interface must be declared in its own file whose name matches the interface's name:

```
// In List.java
public interface List {
    public int size();
    public void add(int x);
    public void insert(int x, int index);
    public void clear();
    public int get(int index), a;
    public int remove(int index);
}
```

Also like a class, the interface declares a type that our local variables can take on:

```
List l;
```

However, unlike a class, we can't instantiate an interface as it is not a class (*i.e.*, blueprint for an actual object):

```
/* Compiler error: List is an interface; it cannot be instantiated. */
List l = new List();
```

This alone is not particularly helpful. We must combine this interface declaration with classes that would like to *implement* the operations defined by this interface. To do this, we note in the relevant class's declaration that it implements the list interface:

```
// In LinkedList.java
public class LinkedList implements List {
    // The usual definition of the linked list class
}
```

Now, the compiler will check the `LinkedList` provides implementations for each of the methods described by the `List` interface. If the `LinkedList` class does not possess one of these methods, then the compiler will raise an error.

By going through this process, the programmer establishes a *subtyping* or *is-a* relationship between the `List` interface and the `LinkedList` class. Because the `LinkedList` class implements the `List`, it must possess all the methods declared by the `List` interface. Therefore, we can use a `LinkedList` in any context where a `List` is expected. In particular, we can assign a `LinkedList` to a `List` variable:

```
List l = new LinkedList();    // Ok!
```

Or we can pass a `LinkedList` to a method expecting a `List`:

```
// Note: the parameter to sumAll is now a List
public int sumAll(List l) {
    int sum = 0;
    for (int i = 0; i < l.size(); i++) {
        sum += l.get(i);
    }
    return sum;
}
// ...
LinkedList l = new LinkedList();
sumAll(l);    // Ok!
```

If our `ArrayList` class also implements the `List` interface:

```
// In ArrayList.java
public class ArrayList implements List {
    //The usual definition of the array list class
}
```

Then we can also pass an `ArrayList` to `sumAll`:

```
ArrayList l = new ArrayList();
sumAll(l);    // Also ok!
```

We say that `sumAll` is *polymorphic* over the type of list it accepts; in fact, it can take any class that implements the `List` interface!

7.1.1 Static and Dynamic Types

The variable declaration:

```
List l = new ArrayList();
```

deserves some study because it is the first instance where the type of the variable does not coincide with the actual type of the object that it is referencing. As a running example, consider the following toy interface and classes:

```java
public interface I {
    public void f();
}

public class A implements I {
    public void f() { System.out.println("A.f"); }
}

public class B implements I {
    public void f() { System.out.println("B.f"); }
    public void g() { System.out.println("B.g"); }
}
```

Before interfaces, the only variable declarations and assignments we could perform were:

```java
A a = new A();  // Ok!
B b = new B();  // Ok!
```

Through `a`, we can call `A`'s `f` method and through `b`, we can call `B`'s `f` and `g` methods. In contrast, we know that the variable assignment `A a = new B();` fails at compile time because `A` and `B` are different types.

With interfaces and subtype polymorphism, we can now perform the following assignments

```java
I i1 = new A();  // Ok!
I i2 = new B();  // Ok!
```

because both `A` and `B` implement the `I` interface. When we call the `f` method on each of `i1` and `i2`, the program produces the following output:

```java
i1.f();   // A.f
i2.f();   // B.f
```

`I` itself does not provide a implementation for `f`; instead, the concrete classes `A` and `B` do so. Therefore, when we invoke `f`, the program uses the type of the *actual object* that the variable references to determine what method should be invoked. For `i1`, the type of object it refers to is `A` and for `i2`, the type of the object it refers to is `B`. We assign the term *dynamic type* to the actual type of the underlying value of a program and the *static type* to the type that the compiler knows about during compilation. The static type for both `i1` and `i2` is `I` which comes from the type of the declared variable.

This seems rather pointless with variable declaration and assignment. However, the nuances become more pronounced when we look at a method:

```java
public static void test1(I i) {
    i.f();
}
```

This method simply calls `f` on its argument, but now it is not clear which `f` will be invoked. Indeed, we don't have enough information to determine this by looking at the method alone as all we know is the static type of `i` (`I`)! We must consider particular method calls and the actual values passed to those calls:

```java
test1(new A())   // a.f()
test1(new B())   // b.f()
```

In summary, when we are checking the type of the variable during compilation, we only consider the static type of the variable. When we perform method lookup, we look at the dynamic type of the object to determine which method to invoke. By doing this, we will never try to call a method (or look up the field) that an object does not possess. To see this, note that the static type only specifies a *subset* of the overall functionality of an object. That is, an object must implement all the methods of an interface it implements but is free to implement others. Therefore, if an object has a particular static type, we know that it at least has the behavior that the static type guarantees.

This leads to one counter-intuitive situation with subtyping

```
I i1 = new A();
I i2 = new B();
i1.g();   // Fails at compilation!
i2.g();   // Also fails at compilation!
```

It makes sense that `i1.g()` should fail because `A` does not provide a `g` method. But `i2.g()` should work because `B` does provide a `g` method. Nevertheless, the compiler rejects both method invocations. Why?

Appealing to our rules about type checking, we note that the compiler uses the *static* type of `i2` during the type checking phase of compilation. Therefore, when checking to see if `i2.g()` is a valid method call at compilation time, the compiler checks to see if `I` has a `g` method, rather than `B`! It does not, so type checking fails. In this sense, type checking is not a *complete* procedure. Type checking will also ensure that a program does not mix up its types. However, type checking will rule out some perfectly sensible programs.

To see why this must be the case, consider a variation on the `test1` method from before:

```
public static void test2(I i) {
    i.g();    // Also fails to compile.
}
```

Unlike the variable declaration case, it is more obvious here that we must rule out `i.g()` because we do not know when type checking the definition of `test2` what values may be passed to it.

7.1.2 Use of the Interface Type

With our toy example, we have two possible choices when declaring a variable that will reference a `B`:

```
I i = new B();
B b = new B();
```

With `b`, we can invoke `B`'s `g` method but with `i`, we cannot. It seems like a no-brainer to always favor the concrete type over the interface type.

However, it turns out that the opposite is true! We want to prefer using the type that is the *most general* type possible that still lets us perform the operations we want on the object. This has two effects:

1. Less importantly, if we need to be more flexible with our types, *e.g.*, we want to pass either a linked list or array list to a method, the interface type accommodates this need.
2. More importantly, if we only need the methods of the interface type and not the additional methods of the concrete type, then using the interface type documents this fact.

Concretely, it is rare that you will need to use a list but then require either linked list- or array list-specific methods. Because of this, you should use the interface type `List`—more generally, the interface type—whenever declaring variables (locals, fields, or parameters) unless your code requires functionality from a particular concrete type.

Exercise (Subtyping as Approximation) Consider the following interface and class declarations:

```java
public interface I { public void foo(); }
public class C implements I {
    public void foo() { System.out.println("C.foo"); }
    public void bar() { System.out.println("C.bar"); }
}
```

And consider the following code snippet:

```java
I i = new C();
i.foo();    // (A)
i.bar();    // (B)
```

For lines (A) and (B) above, answer the following questions:

1. Does the line of code type check?
2. If so, what is the output?

Recall that the static type of `i` is `I` and the dynamic type is `C`. What does this exercise tell you about how *precise* subtyping is? In particular, is there code that is *safe* but *rejected* by Java's type system?

7.2 Parametric Polymorphism

The other pain point concerning our list implementations was the inflexibility surrounding the *carrier type* of the list, *i.e.*, the type of elements the list held. In our original example, we required, *e.g.*, our array list hold integers:

```java
public class ArrayList implements List {
    private int[] data;
    // The rest of the class...
}
```

However, if we wanted to have an array list of some other type, say strings, we needed a whole different class to do this:

```java
public class ArrayStringList implements StringList {
    private String[] data;
    // The rest of the class...
}
```

This class would be a complete duplicate of the original `ArrayList` class except for the carrier type! Furthermore, with our interfaces, we would need to create a new interface type that handles lists of strings. This is clearly an unacceptable situation.

The key insight to solving this problem is realizing that the list doesn't *care* about its carrier type. All of our list implementations don't depend on having an `int` or a `String`. The type might as well be a *variable* that the implementation does not know anything about, just like how a method does not know anything about its parameters (other than its type).

In Java, classes that are parameterized by types in this manner are called *generic classes*. Here is a non-generic version of our linked list's `Node` class specialized to strings:

```java
// In Node.java
public class Node {
    public String value;
    public Node next;
    public Node(String value, Node next) {
        this.value = value;
        this.next = next;
    }
}
```

And here is the generic version of the `Node` class:

```java
// In Node.java
public class Node<T> {
    public T value;
    public Node<T> next;
    public Node(T value, Node<T> next) {
        this.value = value;
        this.next = next;
    }
}
```

Note the differences:

1. Next to the name of the class, we declare the common-separated list of *type parameters* that the class takes in angle brackets. By convention, we usually use the letters `T` and `U` for these type parameters.
2. Where ever we mentioned `String`, we now replace it with our type variable `T`.
3. Where ever we mention the type of `Node`, we must now provide a concrete type to pass along for the class's type parameter. Note that inside the class, the occurrence of `T` is a *use* of the type variable `T`, not a declaration (which occurs at the top of the class).

Now that our node class is generic, we must also make our `LinkedList` list class generic:

```java
// In LinkedList.java
public class LinkedList<T> {
    private Node<T> first;
    public LinkedList() { first = null; }
    public void add(T v) {
        if (first == null) {
            first = new Node<T>(v, null);
        } else {
            Node<T> cur = first;
            while (cur.next != null) { cur = cur.next; }
            cur.next = new Node<T>(v, null);
        }
    }
    // Additional implementation...
}
```

Our generic linked list uses the generic node class, threading along the type variable `T`. Note that when instantiating a generic class, like the `Node`, the class name also includes the instantiating type. This is annoying to do, so Java provides a little amount of *type inference*—inferring the type to pass in based on context. This allows us to instantiate a node with the following syntax instead:

```
first = new Node<>(v, null);
```

where the angle brackets are empty rather than containing the instantiating type.

Now the user of the linked list must specify a concrete type that the linked list ought to hold:

```
LinkedList<String> l1 = new LinkedList<String>();
l1.add("hello");
l1.add("world");
```

The act of passing a concrete type to a generic class is called *instantiating* the class.

By specializing `l` to be a list of strings, we can not add values of other types to this particular linked list. The concrete type only applies to `l`, so we can create another list to hold some other type:

```
  LinkedList<Color> l2 = new LinkedList<Color>();    // Holds Color objects.
```

Finally, we must also make our interface generic to match our `LinkedList`:

```
// In List.java
public interface List<T> {
    public int size();
    public void add(T x);
    public void insert(T x, int index);
    public void clear();
    public T get(int index), a;
    public T remove(int index);
}
```

We then need to change the declaration of our `LinkedList` class thusly:

```
// In LinkedList.java
public class LinkedList<T> implements List<T> {
    // Implementation...
}
```

Note that the first `<T>` (attached to `LinkedList`) is the *declaration* of the type parameter `T`. The second `<T>` (attached to `List`) is a *use* of the type parameter, passing it along to the `List` interface.

7.2.1 Autoboxing and Unboxing

The following code produces a surprising compiler error:

```
List<int> l = new List<>();    // error: cannot instantiate a generic
                               // with a primitive type.
```

It appears that it is impossible to instantiate generics with primitive types, and it is because a generic type parameter must be instantiated to a type variable or a reference type.

Recall that a reference type means that variables of that type are references to objects rather than objects themselves. Because of this, implementing, *e.g.*, a generic field is trivial to do because pointers are the same size no matter what sort of object they are pointing to. This means that the *layout* of the object is identical, irrespective of what the generic type is instantiated to. However, primitives have different sorts of sizes, *e.g.*, a `double` is likely a different size than `int`, and so objects that differ by these types have different layouts.

The workaround is to wrap a primitive in a class. That way, this wrapper can be the type that our generic class holds. Java provides a number of these *wrapper classes* for every primitive:

Type	Wrapper Class
`boolean`	`Boolean`
`byte`	`Byte`
`short`	`Short`
`char`	`Character`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`

(These wrapper classes also provide helper methods for working with these primitive data, *e.g.*, `Integer.parseInt` for parsing a string as an `int`.)

To use these wrapper classes, we instantiate them with a value of an appropriate type which can then be held in our generic list:

```
List<Integer> l = new LinkedList<>();
l.add(new Integer(5));   // Now ok!
int n = l.get(0).intValue();
```

This is also cumbersome to do, so Java provides facilities for automatically putting a primitive type in a wrapper and then unwrapping it when necessary, called *autoboxing* and *unboxing*. With autoboxing, the code to use integers with a generic list is quite natural:

```
List<Integer> l = new LinkedList<>();
l.add(5);            // Auto-boxed int -> Integer
int n = l.get(0);    // Auto-unboxed Integer -> int
```

7.2.2 Generics and Arrays

Finally, there is one additional detail to consider with respect to generics and arrays. Suppose that we were writing our array list class with generics. The following code does not compile!

```
public class ArrayList<T> implements List<T> {
    private T[] data;
    private int last;
    public ArrayList() {
        data = new T[16];    // Error: cannot instantiate generic array!
        last = 0;
    }
    // Rest of implementation...
}
```

It turns out that a (non-)feature of Java generics, *generic type erasure*, does not play well with arrays. Because of this, we cannot instantiate arrays of generic type. To get around this, we must proceed in a roundabout way:

```
public class ArrayList<T> implements List<T> {
    private Object[] data;
    private int last;
    public ArrayList() {
        data = new Object[16];
        last = 0;
    }
    public void add(T v) {
        ensureCapacity();
        data[last++] = v;
    }
    public T get(int n) {
        if (n < 0 || n >= last) { throw new IllegalArgumentException(); }
        @SuppressWarnings("unchecked")
        T ret = (T) data[n];
        return ret;;
    }
    // Rest of implementation...
}
```

Rather than instantiating an array of generic type, we instantiate the array to be an array of type Object. We'll discuss the Object class later in the course, but for now, we can think of Object as a type that everything is an instance of. Because a generic T is an Object, then add can put v in the array directly. However, get must *cast* the result of fetching data from the array since not every object is a T. When performing this cast, the compiler reports a warning saying that the cast is unchecked; we can suppress this warning with an *annotation* above the offending statement. The annotation SuppressWarnings tells the compiler to ignore warnings of a particular type that are generated by the decorated statement.

Exercise (Generalizing Code) Consider the following code snippet:

```
public class Box {
    private int value;
    public Box(int value) { this.value = value; }
    public int getValue() { return value; }
    public void setValue(int value) { this.value = value; }
}
```

Change the definition of `Box` so that it can hold any type rather than just integers.

Chapter 8

Sorting and Searching

One of the essential operations over a data structure is *searching* for a particular value. For example, we can find such a value in an array by checking every element of the array in-order.

```java
/**
 * Performs linear search to find a value in an array of ints.
 * @param arr an array of integers
 * @param v the target value
 * @return the index of the first occurrence of v in the array or -1
 *         if v is not in the array
 */
public static int indexOf(int[] arr, int v) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == v) { return i; }
    }
    return -1;
}
```

This algorithm is appropriately named *linear search* and has worst-case $\mathcal{O}(N)$ time complexity where N is the length of the input array corresponding to checking every element in the array.

While linear search takes advantage of the sequential nature of an array (and in general, any sequential data structure), it does not take *full* advantage of this property. In particular, if the array is *ordered*, then we can employ *binary search* to avoid searching every element of the array.

```java
/**
 * Performs binary search to find a value in a sorted array of ints.
 * @param arr a sorted array of integers
 * @param v the target value
 * @return the index of the first occurrence of v in the array or -1
 *         if v is not in the array
 */
public static int bsearch(int[] arr, int v) {
    return bsearch(arr, v, 0, arr.length);
}
```

```
/**
 * Performs binary search to find a value in a subregion of a sorted array of ints.
 * @param arr a sorted array of integers
 * @param v the target value
 * @param lo the lower bound of the subarray (inclusive)
 * @param hi the upper bound of the subarray (exclusive)
 * @return the index of the first occurrence of v in the array or -1
 *         if v is not in the array
 */
public static int bsearch(int[] arr, int v, int lo, int hi) {
    if (lo > hi) {
        return -1;
    } else {
        mid = lo + (hi - lo) / 2;
        if (v < arr[mid]) {
            return bsearch(arr, v, lo, mid);
        } else if (v > arr[mid]) {
          return bsearch(arr, v, mid + 1, hi);
        } else {
            return mid;
        }
    }
}
```

As we discussed earlier, binary search has $\mathcal{O}(N \log N)$ worst-case time complexity by repeatedly chopping the array of the array under consideration in half. The downside, of course, is that the input array must be sorted. However, since searching is such a fundamental operation over a data structure, it is frequently more efficient to sort our data before we analyze it.

In this chapter, we investigate fundamental sorting algorithms over sequential data structures. There are many such sorting algorithms with unique trade-offs between each other in terms of performance and behavior.

8.1 Quadratic Sorts

One natural way we can deriving sorting algorithms is to employ *invariant design* directly on the traversal of an array. It makes sense to attempt to traverse a list left-to-right to try to place each element in its sorted position. Depending on the invariant we place on this traversal, we obtain different sorting algorithms.

For example, consider the following invariant:

Invariant 1. *When we are at position i in the array, the i smallest elements are to the left of the array.*

Pictorally, we represent this invariant in an array as follows:

```
[///////////|           ]
  smallest   ^
    elts     i
   sorted
```

To implement this sorting algorithm, we need to implement a loop that satisfies the invariant:

```
for (int i = 0; i < arr.length; i++) {
    // Ensure that the i smallest elements of the array
    // are in the indices 0, ..., i
}
```

This requires that we find the smallest element in unsorted region of the array and then place it at position *i*—the smallest element in the unsorted region becomes the largest element in the sorted region.

```
public static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int lowestIndex = i;
        for (int j = i; j < arr.length; j++) {
            if (arr[j] < arr[lowestIndex]) {
                lowestIndex = j;
            }
        }
        swap(i, lowestIndex);
    }
}
```

Note that we do not need to consider the final element in the outer loop because we know by our invariant that it must be the largest element in the array by virtue of being the only one in the unsorted region. This sorting algorithm is called *selection sort*.

For example, consider the following array $[3, 1, 5, 9, 4]$. Here is an example step-by-step execution of selection sort on the array:

$$[\underline{3}, 1, 5, 9, 4]$$
$$[1, \underline{3}, 5, 9, 4]$$
$$[1, 3, \underline{5}, 9, 4]$$
$$[1, 3, 4, \underline{9}, 5]$$
$$[1, 3, 4, 9, 5]$$

The underline value represents i before the execution of each inner loop of the algorithm.

To analyze the runtime of this sorting algorithm, we note that the first iteration of the outer loop runs for n iterations, the second iteration $n - 1$ times, the third iteration $n - 2$, and so forth. Inside each inner loop, we perform a constant number of array accesses. Afterwards, we perform an array swap. Thus, the time complexity runtime is $\mathcal{O}(n^2)$. Note that this is irrespective of the shape of the array so it is the best-, worst-, and average-case runtimes.

Can we do better? What if we *relaxed* the invariant so that rather than demanding that the sorted region contain the smallest elements of the array, it simply contains sorted elements. By doing so, we still obtain a sorted array in the end, but also gain some flexibility in our algorithm that we can use to make it more efficient in some cases.

Let's try working with the following invariant:

Invariant 2. *When we are at position i in the array, the elements up to i are in sorted order.*

Pictorally, the invariant looks as follows:

```
[//////////|          ]
   sorted   ^
            i
```

Intuitively, by working with the element at i rather than seeking the next smallest element in the unsorted portion of the array, we cut down on the amount of array accesses we make. For example, consider sorting the same array $[3, 1, 5, 9, 4]$ using this technique:

$$[3, \underline{1}, 5, 9, 4]$$
$$[1, 3, \underline{5}, 9, 4]$$
$$[1, 3, 5, \underline{9}, 4]$$
$$[1, 3, 5, 9, \underline{4}]$$
$$[1, 3, 5, \underline{4}, 9]$$
$$[1, 3, 4, 5, 9]$$

For each element we encounter (underlined above), we *push* it into its proper position in the sorted region of the array. We accomplish this by swapping that element with the element to its left whenever they are out of order. For example, at the first step of the algorithm, we swap 3 and 1 because they are out of order. Likewise, to put 4 in its proper position, we swap it several times to get into the 2nd index of the array.

With this, we can implement this sort, called *insertion sort*, as follows:

```java
public static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        for (int j = i; j > 0 && arr[j-1] > arr[j]; j--) {
            swap(arr, j, j-1);
        }
    }
}
```

Rather than seeking for the smallest element in the unsorted region of the array, the inner for-loop swaps element i downward into the sorted region until it is in its sorted position.

Like selection sort, insertion sort exhibits a doubly-nested for-loop over the lenghth of the input array which, at first glance, implies that its runtime complexity is $\mathcal{O}(n)$. However, the behavior of the inner for-loop depends on the shape of the array. For example, if the array is already sorted, then every iteration of the inner for-loop immediately terminates because for any j, `arr[j-1] < arr[j]`. This means that in this best case, the runtime of insertion sort is $\mathcal{O}(n)$. Sorting an already-sorted array may feel like an unlikely corner case. However, in practice this happens with some frequency because sometimes our data already comes sorted and it already costs $\mathcal{O}(n)$ time to check that this is the case.

In contrast, the worst case is when the array is in *reverse sorted* order. When this is true, every j encountered in the inner-for loop is the smallest element to be inserted into the sorted region of the array. Thus, every run of the inner-for loop performs $j - 1$ swaps to get the jth element to the front of the array. Because j ranges from 1 to `arr.length`, this means that in the worst case, insertion sort still performs in $\mathcal{O}(n^2)$ time. In practice, insertion sort performs swaps at each run of the inner-loop proportional to the length of the array, so the average case runtime is also $\mathcal{O}(n^2)$.

8.2 Merge Sort

There are many other quadratic sorts to consider. However, they all have the same $\mathcal{O}(n^2)$ performance, and most have worse properties than insertion sort, which will be our go-to quadratic sort. Can we do better than $\mathcal{O}(n^2)$? It turns out we can by employing the same *divide-and-conquer* technique that we employed to implement binary search.

Suppose that we are able to sort two halves of an array.

```
[////////////][\\\\\\\\\\\\]
    sorted        sorted
```

Can we combine them into a sorted whole? We can do this through a process called *merging*. Consider a concrete array $[1, 4, 9, 2, 3, 8]$ where the first three elements are sorted and the last three elements are sorted. To merge these two sub-arrays into a sorted whole, we first create an auxiliary array the same size as the input array that holds our sorted result. We then start with two pointers into the array, each at the start of the two sub-arrays.

$$[\underline{1}, 4, 9, \underline{2}, 3, 8] \longrightarrow [\underline{0}, 0, 0, 0, 0, 0]$$

We maintain the invariants that:

(a) The two pointers in the input array point to the smallest elements in each of the sub-arrays that have not yet been sorted.

(b) The elements to the left of the pointer in the auxiliary array are the smallest elements of the input array in sorted order.

At the end of the merge operation, the auxiliary array contains all of the elements of the input array in sorted order. We can then copy these elements back into the input array as our final result.

To do this, at every step of the merge operation, we take the minimum of the two elements pointed at in our input array and then copy it into the current position of the auxiliary array. We then advance the pointer that pointed to the element as well as the auxiliary array's pointer. Here's an example run of the merge operation on the array above:

$$\begin{aligned}
[\underline{1}, 4, 9, \underline{2}, 3, 8] &\longrightarrow [\underline{0}, 0, 0, 0, 0, 0] \\
[1, \underline{4}, 9, \underline{2}, 3, 8] &\longrightarrow [1, \underline{0}, 0, 0, 0, 0] \\
[1, \underline{4}, 9, 2, \underline{3}, 8] &\longrightarrow [1, 2, \underline{0}, 0, 0, 0] \\
[1, \underline{4}, 9, 2, 3, \underline{8}] &\longrightarrow [1, 2, 3, \underline{0}, 0, 0] \\
[1, 4, \underline{9}, 2, 3, \underline{8}] &\longrightarrow [1, 2, 3, 4, \underline{0}, 0] \\
[1, 4, \underline{9}, 2, 3, 8] &\longrightarrow [1, 2, 3, 4, 8, \underline{0}] \\
[1, 4, 9, 2, 3, 8] &\longrightarrow [1, 2, 3, 4, 8, 9]
\end{aligned}$$

Note that we stop comparing sides once we reach the end of one of the sub-arrays. In the case above, we reached the end of the right-hand sub-array first. The remaining elements of the left-hand sub-array are then copied into the auxiliary array—they are the largest elements of the overall array, already in order.

With the merge operation defined, we can define *merge sort* over a sub-array of the overall input array as follows:

- If the sub-array to sort is size 1 or less, then it is already trivally sorted.
- Otherwise, recursively sort the left- and right-hand halves of the sub-array and then merge the two into a final sorted whole.

To analyze the time complexity of merge sort, we first analyze the runtime of the merge operation. Given a sub-array of length n, the merge operation touches every element of that sub-array twice—once to copy it into the auxiliary array and once to copy it back in sorted order after-the-fact. This happens irrespective of the composition of the sub-array, so the merge operation performs $2n$ operations.

Now, let's consider the runtime of the complete sorting algorithm. It is recursive, and thus requires a recurrence relation $T(n)$ describing the runtime of the algorithm over the length of the array n. The base case is when the array has at most one element where we immediately return. In the recursive case, we recursively sort both halves of the input array and perform merge in $2n$ steps. This yields the following recurrence:

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

We can derive a closed-form solution for this recurrence by rewriting the formula in terms of k unfolding of the recurrence and setting k equal to the case when the recurrence bottoms out. After k unfoldings, the recurrence yields:

$$\begin{aligned}T(n) &= 2n + 2T(\frac{n}{2})\\ &= 2n + 2(2\frac{n}{2} + 2T(\frac{n}{4})\\ &= 2n + 2n + 4T(\frac{n}{4})\\ &= 2n + 2n + 2(4\frac{n}{4} + 2T(\frac{n}{8}))\\ &= 2n + 2n + 2n + 2T(\frac{n}{8})\\ &= k2n + 2^kT(\frac{n}{2^k})\end{aligned}$$

The recurrence bottoms-out when $\frac{n}{2^k} = 1$. Solving in terms of n, this occurs when $k = \log_2 n$. Plugging back into the recurrence yields:

$$\begin{aligned}T(n) &= 2n\log_2 n + 2^{\log_2 n}T(\frac{n}{2^{\log_2 n}})\\ &= 2n\log_2 n\end{aligned}$$

Thus, $T \in \mathcal{O}(n \log n)$.

Similarly, let's consider the space complexity of merge sort. Every call to merge allocates a separate array the same size as the input. This yields the following recurrence for the space complexity:

$$S(1) = 0$$
$$S(n) = n + 2S(\frac{n}{2})$$

Performing a similar analysis above yields that $S \in \mathcal{O}(n \log n)$. However, we can do better than this by simply passing in a single array the size of the overall input array to merge sort once and perform all our scratch work in that array. This requires a single allocation of a n-length array upfront which makes the space complexity $S \in \mathcal{O}(n)$.

8.3 Quicksort

Can we do better than merge sort? The runtime is nice, but can we get rid of that $\mathcal{O}(n)$ space complexity? To do this, let's keep the "divide-and-conquer" style of behavior (as this is what nets us that nice linearithmic time complexity), but vary other aspects of the algorithm. Summarizing merge sort's behavior:

1. Find the midpoint of the array.
2. Recursively sort the two halves of the array.
3. Merge the two sorted halves into a completely sorted array.

Note that we recursively sort and then merge. What if we flipped this around, performing the actual "sorting" behavior first, and then recursively sorted the halves? This idea forms the basis of the final sorting algorithm that we will cover: *Quicksort*.

Let's first imagining choosing the midpoint of the array. We cannot merge the two halves of the array at this point because they are unsorted. But what we can we do to make progress? We can *partition* the array based off of the value at the midpoint of the array. If our array has this shape beforehand:

```
[        ][ ][        ]
           mid
```

We want to modify it so that it has the following shape:

```
[ <mid ][ ][ mid> ]
          mid
```

That is, all the elements less than `mid` are to the left and all the elements greater than `mid` are to the right. This way, (1) we know `mid` is in the correct position and (2) we can recursively sort the left- and right-hands of the array independently. Note that the left- and right-hand partitions of the array may not be sorted yet—the goal of our recursive calls are to sort these halves.

How do we perform this partition operation? Let's imagine how we might do this on the following array:

$$[3, 9, 2, 8, 6, 4, 1, 7, 5]$$

The midpoint is the element 6. How do we partition this array? We can *swap* elements to the left of the midpoint with an element to the right of the midpoint if they are not in the correct position; for example, 1 and 9 ought to be swapped with respect to the pivot, 6. However, by doing so, the resulting left-hand and right-hand sides may not be equal in size. For example, if we partition the following array:

$$[1, 1, 7, 1, 6, 1, 1, 1, 1]$$

Then the result should be:

$$[1, 1, 1, 1, 1, 1, 1, 6, 7]$$

The left-hand partition is far larger than than the right-hand partition because only one element in the array is greater than 6.

Therefore, we cannot simply leave the midpoint where it is as it might move as a result of the partition. To alleviate this, we first move the midpoint to the end of the array via a swap:

$$[3, 9, 2, 8, 5, 4, 1, 7, 6]$$

And then we proceed to swap elements with our two fingered approach similar to the merge operation, except that left-hand finger starts on index 0 and the right-hand finger starts on the second-to-last element. Each finger then searches for the next element on its respective side to swap, and then we swap the elements. The procedure ends when our fingers cross. Here's an example execution of the procedure using our original array:

$$[\underline{3}, 9, 2, 8, 5, 4, 1, \underline{7}, 6]$$
$$[3, \underline{9}, 2, 8, 5, 4, 1, \underline{7}, 6]$$
$$[3, \underline{9}, 2, 8, 5, 4, \underline{1}, 7, 6]$$
$$[3, \underline{1}, 2, 8, 5, 4, \underline{9}, 7, 6]$$
$$[3, 1, 2, \underline{8}, 5, 4, \underline{9}, 7, 6]$$
$$[3, 1, 2, \underline{8}, 5, \underline{4}, 9, 7, 6]$$
$$[3, 1, 2, \underline{4}, 5, \underline{8}, 9, 7, 6]$$
$$[3, 1, 2, 4, \underline{5}, \underline{8}, 9, 7, 6]$$
$$[3, 1, 2, 4, 5, \underline{8}, 9, 7, 6]$$

After this swapping process, we know that everything to the left our fingers is less than the midpoint and everything to the right of our fingers is greater than the midpoint. We can then swap the midpoint back in where our pointers crossed as our pointers now point to an element larger than the midpoint.

```
[3, 1, 2, 4, 5, 6, 9, 7, 8]
 /////////////     \\\\\\\
      <6              >6
```

Note that the fact that 6 was the midpoint of the array did not matter in the end. Indeed, we could have chosen any element to serve as the *pivot* of the partition operation. And as you might guess, our choice of pivot heavily influences the runtime of quicksort. In particular, we want to choose pivots such that we partition in the array to obtain subarrays of size $n/2$. Poor choices of pivots result in dividing up the array so that one array has size 1 and the other than size $N - 1$.

Given that we have some way to choose a good pivot, quicksort proceeds as follows:

1. Choose a pivot in the array.

2. *Partition* the array into two subarrays using the pivot.
3. Recursively quicksort the two subarrays.

As noted above, our choice of pivot heavily influences the complexity of quicksort. In the best case, our choices of pivots allows us to evenly partition the array in half. From our description of the partition operation above, we see that we walk the array with our two fingers, so it takes linear time. Overall, this means that the recurrence for quicksort in the best case is $T(n) = n + 2T(n/2)$ which we know from mergesort means that the algorithm takes $\mathcal{O}(n \log n)$ time in the best case.

In the worst case, the pivot divides up the array so that we have a subarray of size 1 and a subarray of size $n - 1$. The recurrence here is now $T(n) = n + T(n-1) + T(1)$. The base case of quicksort, like mergesort, is trivial so $T(1)$ is just 1. Expanding the recurrence for a pattern yields:

$$\begin{aligned} T(n) &= n + 1 + T(n-1) \\ &= n + 1 + ((n-1) + T(n-1) \\ &= 2n + 1 + T(n-2) \\ &= 3n + T(n-3) \end{aligned}$$

After k expansions, we have $T(n) = kn + 1 - k + T(n-k)$. This process bottoms out when $n = k$ where we have $T(n) = n^2 + 1 - n + 1 = n^2 - n + 2$. The quadratic term dominates so we have $T \in \mathcal{O}(n^2)$.

On average, we divide up the array by some fraction—not necessarily $n/2$ but say, $n/4$ and $3n/4$ which results in a linearithmic ($\mathcal{O}(n \log n)$) runtime.

This seems worse than mergesort which has a consistent $\mathcal{O}(n \log n)$ runtime. However, unlike mergesort, quicksort takes $\mathcal{O}(\log n)$ space due to recursive function calls; it does not allocate extra anything on the heap.

So how do we choose the pivot? Ideally, we would like to choose the median of the array which guarantees that we partition the array optimally. However, because the array is unsorted, finding the median takes $\mathcal{O}(n^2)$ time which would ruin the runtime of the sort. Thus, we need a pivot-choosing method that is (1) simple, (2) approximates the best-case scenario of the median, and (3) avoids the worst-case scenario of choosing the smallest or largest element.

A simple choice is the first element of the array. However, if the array is already sorted, then this element is the smallest, leading to the worst-case behavior described above. Another choice is to simply choose a random element. This makes it unlikely to fall into the worst case scenario, but the behavior of mergesort then becomes more unpredictable.

Our original choose of the midpoint of the array is actually somewhat sensible. In the case when the array is sorted, we choose the optimal value. However, this choice does not approximate the median value at all when the array is unsorted.

A better choice that avoids the worst case and approximates the median is the *median-of-three* rule. The median of three rule says the following:

> Choose the pivot to be the median of the values found in the first, middle, and last positions of the array.

Note that we can perform this pivot choice operation in constant time as it is just a small number of comparisons between the first, middle, and last values. This is strictly better than simply choosing the midpoint—in the case where the array is sorted, we correctly choose the middle value. In the case where the array is unsorted, we at least get the median of some range of numbers. We might get unlucky and choose the three smallest numbers, but at least we are *guaranteed* to never fall into the absolute worst case scenario of choosing the smallest or largest values of the array.

8.4 Generic Constraints

So far, you have likely written your sorting routines in terms of integer arrays. However, we would like to sort arrays of any type without having to duplicate code. To do this in Java, we use generics:

```java
public static <T> void insertionSort(T[] arr) {
    for (int i = 0; i < arr.length; i++) {
        for (int j = i; j < arr.length; j++) {
            if (arr[j] < arr[i]) {
                T temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}
```

Like classes, we can declare a static method as generic by introducing a type parameter enclosed in angle brackets. For a method, the type parameter declaration goes before the return type (above, it is <T>). Alternatively, we can also enclose the method in a generic class and use the type parameter from the class declaration. (Note that instance methods cannot be generic—their enclosing class must be declared generic instead.)

However, this method as-is does not compile. Java gives the following error for the above code snippet:

```
Sort.java:5: error: bad operand types for binary operator '<'
                if (arr[j] < arr[i]) {
                           ^
  first type:  T
  second type: T
  where T is a type-variable:
    T extends Object declared in method <T>insertionSort(T[])
1 error
```

The problem stems from the fact that `T` is a generic type. The only operations we can perform on a `T` are those guaranteed by the `Object` class (since `Object` is the root of the Java class hierarchy). Comparison (*i.e.*, the less-than operator) is not one of these universal operations. And indeed, some types may not have the less-than operator defined for them.

We need some mechanism for *constraining* instances of this generic type to be only those that have some suitable definition of comparison. We do this through *generic type constraints.*

```java
public static <T extends Comparable<T>> void insertionSort(T[] arr) {
    for (int i = 0; i < arr.length; i++) {
        for (int j = i; j < arr.length; j++) {
            if (arr[j].compareTo(arr[i]) < 0) {
                T temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}
```

Here, we augment the type parameter declaration to be a type parameter called T that *must implement the Comparable<T> interface*. Comparable<T> is an interface in the Java standard library that objects implement to define what it means to compare them to other objects of the type parameter. Here it the complete definition of the interface:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

Implementors provide a method, compareTo, that takes in an object of the given generic type and produces the following return value:

- A negative integer if this object is "less than" the other object.
- 0 if this object is "equal" to the other object.
- A positive integer if this object is "greater than" the other object.

Note how the compareTo method supercedes the funcitonality of the less-than, greater-than, and equals operators. So rather than performing a less-than check in insertion sort, we use compareTo to determine if the given element is the new minimum.

To use insertionSort now, we must keep in mind that it requires an array of reference type. This means we can no longer pass in an int array; we must pass in an Integer array instead:

```
// NOTE: this works because Java auto-boxes the ints into Integer objects...
Integer[] arr = new Integer[] { 4, 6, 1, 2, 9, 9, 10, 4, 2 };
insertionSort(arr);
```

The Comparable interface allows us to state what it means to compare any two types of objects. In other words, it allows us to impose an *ordering* on these types of objects. Most commonly, we're interested in specifying an ordering on objects of the *same type*, for example ordering dogs by age:

```
public class Dog implements Comparable<Dog> {
    private int age;
    // ...
    @Override
    public int compareTo(Dog other) {
        if (this.age < other.age) {
            return -1;
        } else if (this.age > other.age) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

The Java standard library defines what it means for all the wrapper classes to be compared to each other (*e.g.*, `Integer`, `Double`) as well as common classes such as `String`. But we are free to define what the ordering is between types and also define what combinations of types have an ordering.

8.4.1 Wildcards and Type Bounds*

Our `insertionSort` method is close to being perfect, but not quite satisfactory. The generic type constraint requires that the object provide an implementation for being `Comparable` to objects of its own type. But we ought to be more flexible than this. Note that we use the `Comparable` interface to compare elements of the same type. If our object provides an implementation of `Comparable<T>` where `T` is a *superclass* of the object, then we can still use this compareTo method to compare elements in the array. For example, if we have:

```java
public class Dog implements Comparable<Animal> { /* ... */ }
```

The `Dog` class has defined what it means for itself to be compared to other animals. However, the `Dog` is an `Animal`, so we can compare two dogs against each other (although we are unable to use the `Dog`-specific parts of the other `Dog` object in the comparison). We can reflect this desire—we want types that can be compared with each other or themselves and a superclass—with a *generic wildcard*.

```java
public static <T extends Comparable<? super T>> void insertionSort(T[] arr) {
    for (int i = 0; i < arr.length; i++) {
        for (int j = i; j < arr.length; j++) {
            if (arr[j].compareTo(arr[i]) < 0) {
                T temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}
```

The `? super T` clause introduces a *wildcard* (denoted by the `?`) and puts a *lower bound* on that wildcard. The effect is that we are further restricted in the type that is used used to instantiate the Comparable interface. The method takes an array whose carrier type `T` implements the Comparable interface against types that are `T` or a superclass of `T`.

Like subtyping of generic type parameters, wildcards and bounds get very tricky. Here are your options:

- `?` specifies an unconstrained wildcard generic type. Such a generic type can be instantiated with any type.
- `? super T` specifies a wildcard generic type with an *lower bound* of `T`. Such a generic type must be instantiated with `T` or a superclass of `T`.
- `? extends T` specifies a wildcard generic type with a *upper bound* of `T`. Such a generic type must be instantiated with `T` or one of its subclasses.

To deterimine which of these wildcards to use in the arguments of methods that we write, we must consider how the argument to the method is used:

- Does the method use the parameter as an data source? We call such a parameter an *in parameter*.
- Does the method use the parameter as a data sink or output? We call such a parameter an *out parameter*.

For our sorting algorithms above, while the input array is an "input", we are really changing the structure of the array (by sorting it). Therefore, the array functions as an out parameter. Once we have identified for each parameter whether it is an in or out parameter, then for type parameters found in the parameter's corresponding type, we choose the following:

- For *in parameters*, we use an upper bound wildcard: `? extends T`.
- If the *in parameter* does not have any particular type constraints, *i.e.*, we only use it as if it was an `Object`, then use an unbounded wildcard: `?`.
- For *out parameters*, we use a lower bound wildcard: `? super T`.
- For parameters that we consider both *in* and *out*, we do not use a wildcard. Instead we use a concrete type instead.

As an example, consider a method that performs a generic copy from a source List to a destination List.

```
public static <T> void copy(List<? extends T> src, List<? super T> dst) { /* ... */ }
```

The `src` parameter is the source of the copy, so it is an *in parameter*. In contrast, the `dst` parameter is the destination of the copy, so it is an *out parameter*.

Chapter 9

Higher-Order Sequential Operations

Many of the operations we wish to perform over lists have common structure. In this chapter, we investigate the most common of these patterns and how we can abstract them into powerful, generic list manipulation algorithms.

9.1 Iteration

The most common operation we perform over a List is the *traversal*. We walk the list, doing something to each element, *e.g.*, finding the largest element, summing up all the elements, or transforming all the elements. To do this, we normally write a for-loop:

```
List<T> list = /* ... */;
for (int i = 0; i < list.size(); i++) {
    ... list.get(i) ...
}
```

If our list is an `ArrayList<T>`, then this loop has linear time complexity because we perform a constant-time operation, `get`, on each element of the list. However, if the list is `LinkedList<T>`, then this loop has quadratic time complexity because `get` now takes linear time.

This additional overhead for linked list traversal seems unnecessary, however. We know that we can implement such a traversal in linear time by performing a standard linked list traversal and holding onto the *current node* rather than the *current index*. However, the `Node` class is inaccessible outside of the `LinkedList` class because we designed the `Node` class to be `private` and nested within the `LinkedList` class[1] We need some mechanism for *abstracting* the method by which we traverse a particular list in an efficient manner, whether that list is an array list or linked list. This mechanism is called *iteration* and in Java we use an `Iterator` object to perform the traversal.

We can think of an iterator as a *cursor* into our list, maintaining our current position as we iterate over the structure. Initially, the iterator points to the beginning of the list, and we can fetch the element that the cursor currently references. The `List` interface of the Java standard library provides a method, `iterator()`, that produces such an iterator object. Here is an example of using an iterator object to walk a `List`:

[1]It is important to note that hiding the `Node` class from clients of the `LinkedList` class is not a hard-and-fast rule; it is a design choice. Some libraries such as the .NET library (the standard library of the C# programming language) expose its `Node` class to clients, trading off uniformity and simplicity of interface for flexibility in manipulating the structure of the linked list.

```
List<String> list = /* ... */;
Iterator<String> iter = list.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

The `Iterator` itself is an interface with two critical operations:

- `boolean hasNext()`: returns `true` if the cursor is pointing to an element of the list, *i.e.*, there are still elements left to process.
- `T next()`: retrieves the element currently under the cursor *and* advances the cursor one position.

The `ArrayList` and `LinkedList` provide two implementations of this interface:[2]

- The array list's iterator object maintains the current index as its cursor. `hasNext` checks to see if the index exceeds the list's size. `next` retrives the element at the current index and increments the index afterwards.
- The linked list's iterator object maintains a pointer to the current node as its curosr. `hasNext` checks to see if the pointer is pointing to `null`. `next` retrives the element stored in the node currently being pointed to and advancing the pointer afterwards.

Because of this, the list iteration code above using an `iterator` operates in linear time irrespective of the type of list being used!

Trade-offs By using the `Iterator` class, we can write code that efficiently traverses a list that operates uniformly over both array lists and linked lists. In addition to this solution being more concise than a traditional for-loop, less can "go wrong" with this code. There are no indices or references to worry about—indeed, the iterator object provides exactly the set of operations we need to perform a forward traversal and nothing more. However, in return, note that the iterator interface constrains us to only move forward through a list. In particular, we cannot:

- Go backwards in the list.
- Go through two lists *pairwise*, *i.e.*, look at the first elements from each list together, then the second elements, and so forth.
- Delete elements from the list.

While other kinds of iterators that Java provide backwards movement (*e.g.*, the `ListIterator` class), in general we cannot capture all of the patterns of iteration or behavior we might want to perform. However, basic iteration is so common when using sequential structures (indeed, any data structure), that the behavior that an iterator provides is still useful even though it is highly specialized.

[2] The Java `Iterator` interface specifies a third, optional method, `remove()`, which removes the element currently being pointed at by the iterator. Removal is considered optional because some data structures may not support removal. Java's list data structures (`ArrayList` and `LinkedList`) support `remove`, but we might want to iterate over something whose elements cannot be directly modified, *e.g.*, an immutable list. In this case, `remove` should throw an `UnsupportedOperationException`.

9.1.1 Iterator Implementation

How is the `Iterator` interface implemented? As described above, the `ArrayList` and `LinkedList` classes each provide their own version of the `Iterator` interface. We could perform the iteration using methods of the `List` interface, but as we noted at the top of this reading, this makes linked list traversal quadratic instead of linear. Therefore, we must implement the `Iterator` interface differently for `ArrayList` and `LinkedList`.

Array Lists To see how to implement the `Iterator` interface for each of `ArrayList` and `LinkedList`, it is useful to appeal to the code for traversing these structures. For example, recall that an array list maintains two fields—`size` and `data`. Using these fields, we can write code to traverse the elements of the list as follows:

```
int index = 0;
while (index < size) {
    ... data[index] ...
    index++;
}
```

(Note that we iterate up to the size of the list rather than the length of the array because the elements of the list are stored at indices `0` to `(size - 1)`.) To translate this traversal operation into a class, we must recognize the relevant *state* of the operation and how the operation maps onto the iterator interface's methods, `hasNext` and `next`.

For array list traversal, the state of its traversal is the `index` being incremented in the loop. In terms of this state, `hasNext` is equivalent to the guard in the for-loop and `next` is a combination of the array access inside of the loop body and the increment of `index`. Thus, we can implement the `Iterator` interface for the `ArrayList` class as follows:

```
public class ArrayList<T> {
    private int size;
    private Object[] data;
    // ...
    private class ArrayListIterator implements Iterator<T> {
        private int index;
        public ArrayListIterator() {
            index = 0;
        }
        public boolean hasNext() { return index < size; }
        public T next() { return data[index++]; }
    }

    public Iterator<T> iterator() {
        return new ArrayListIterator();
    }
}
```

The `ArrayListIterator` class implements the array list traversal-specific behavior described above. With this class, we make two important design decisions which make it an air-tight abstraction:

- It is contained within the `ArrayList` class so that it has access to an array list object's fields.
- It is marked `private` because clients of the array list class should not be aware of the `ArrayListIterator` class.

The fact that the `ArrayListIterator` is hidden from clients seems counter-intuitive. The whole point of the iterator is to give clients a way to traverse the data structure!

The way we expose this class to users is through the `iterator()` method of the `ArrayList` class which creates an `ArrayListIterator` and immediately returns it to the client. Note that the `iterator()` method returns a value of the more general type `Iterator`, rather than the actual type of the value, `ArrayListIterator`. This is sound because `ArrayListIterator` implements `Iterator`, so it provides all of the behavior expected of any `Iterator` object. But by using this particular arrangement, we hide the implementation details of the array list class—namely that it implements its iteration behavior with the `ArrayListIterator` class—from clients but still expose an iterator for external use.

Linked Lists Recall that the `LinkedList` class contains a field `first` that is a reference to the first `Node` in the linked list. The `Node` class itself has a `value`—an element of the list—and a `next` field—a reference to the next node in the list. With this in mind, we can perform linked list traversal using the following code skeleton:

```
Node<T> cur = first;
while (cur != null) {
    ... cur.value ...
    cur = cur.next;
}
```

Unlike the array list, we must keep track of the current *node* that we are on and advance the iterator by advancing our "current" reference. Otherwise, we'll use the same set up of an inner iterator class exposed with an `iterator()` method that produces an instance of this class and gives it to the user:

```
public class LinkedList<T> {
    private static class Node<T> { /* ... */ }
    private Node<T> first;
    // ...
    private class LinkedListIterator implements Iterator<T> {
        private Node<T> cur;
        public LinkedListIterator() { cur = first; }
        public boolean hasNext() { cur != null; }
        public T next() {
            T ret = cur.value;
            cur = cur.next;
            return ret;
        }
    }

    public Iterator<T> iterator() {
        return new LinkedListIterator();
    }
}
```

Note that the `next()` method of the `LinkedListIterator` class operates in constant ($\mathcal{O}(1)$) time. This allows us to iterate over a linked list in linear ($\mathcal{O}(n)$) time similarly to an array list.

9.1.2 For-each Loops

Because traversal with an iterator is so common in Java (and in general, programming), the language provides special syntax for this through the *for-each* loop:

```
List<String> list = /* ... */;
for (String str : list) {
    ... str ...
}
```

A for-each loop is similar in syntax to a for-loop. However, a for-each loop only has two components inside of the parenthesis:

- The variable that each element of the list is bound to while the loop runs (`str` in the above example).
- The actual list we are traversing over (`list` in the above example).

The above for-each loop is equivalent to the following while-loop:

```
List<String> list = /* ... */;
Iterator<String> iter = list.iterator();
while (iter.hasNext()) {
    String str = iter.next();
    ... str ...
}
```

In fact, when you use a for-each loop, the Java compiler replaces it with this while-loop! Such a programming language construct that acts as "shorthand" for some other combination of programming language constructs (rather than doing that cannot otherwise be replicated in the language) is called *syntactic sugar*). Thus, we say that a for-each loop is syntactic sugar for using the `Iterator` class to perform a traversal.

This behavior is not specific to lists; we can use a for-each loop to iterate over antything produces an `Iterator` object! The Java compiler checks for this by seeing if a class implements the `Iterable` interface. The `Iterable` interface requires one method of its implementors:

- `Iterator<T> iterator()`: returns an iterator to the "beginning" of this object.

This is precisely the method we use from the `List` interface to grab our iterators. If you want your own classes to be usable in a for-each loop, then you must:

- Create a structure-specific class that implements the `Iterator` interface with appropriate traversal behavior for that data structure.
- Have the data structure class implement the `Iterable` interface. The `iterator()` method you define should return an instance of the class's corresponding `Iterator` sub-class.

As a final note, it is worthwhile to realize that iteration is a general concept that applies to any sort of data structure we may encounter, not just lists. If we think of our data structures as *containers* for data, we will frequently want to access all of the data stored in a particular container. While sequential structures provide a very natural implementation of iteration based on the order of their elements, we can extend this notion to the hierarchical and mapping structures that we will encounter later.

9.2 Infinite Sequences

So far, we have studied sequential structures of *fixed size*—arrays—and *variable size*—lists. Note that these variable-sized lists, while possessing the capacity to grow, are ultimately *finite* in length. That is, the size of the list is always some finite value although it can grow as necessary. Does it make sense to study sequences with a potentially *infinite* number of values? At first glance, such a structure seems unusable! In the previous section, we identified that the most common operation over a sequence is the traversal, but how can we traverse a structure with an infinite number of values?

Indeed, this is impossible to do as such as a traversal would never end; the best we can do is sample finite prefixes of the infinite sequence What we'll do instead is rather than traversing the structure repeatedly to perform operations over it, we'll build up a collection of computations that will be performed only when we ask for elements from the stream. This *lazy model of computation* where our transformations over the data do not occur until it is necessary is not just useful for infinite sequences, they are also useful for finite sequences as well.

9.2.1 Streams

A stream is a potentially infinite sequence of values. For example, network traffic from the Internet or input from the user can be thought of as a stream because we do not know if the data from the source (the Internet or the user) will ever end. Finite sequences can naturally be thought of as a stream as well, *i.e.*, lists or files. In particular, very, very large sequences of data, say in the gigabytes or terabytes, are effectively infinite streams as we cannot practically traverse them using standard list operations.

In Java 8, the `stream` package contains a number of classes to create and manipulate streams. In particular, the `Stream` class represents this abstract data type of an infinite sequence. As a example, suppose that we wish to read in data from a file as a `Stream` object. To represent this data as a `Stream`, we have two choices: (1) read in the data into a list and then make a stream out of the list or (2) read in the data as a stream directly.

The first approach should be familiar to you. In Java, we use the `Scanner` class to read in a file line by line, word by word, *etc.*, and then place that data into a list. From there, we can use the `stream()` method of the `List` class to create a `Stream` object that reads from the list.

```
// throws FileNotFoundException
Scanner src = new Scanner(new File("data.txt"));
List<String> data = new LinkedList<>();
while (src.hasNextLine()) {
    // Assuming that one datum appears per line in the file...
    data.add(src.nextLine());
}
Stream<String> stream = data.stream();
```

Note that the `Stream` class, like the `List`, is generic in the carrier type of the `Stream`.

The alternative approach uses additional Java 8 functionality to perform this line-by-line reading concisely:

```
// throws IOException
Stream stream = Files.lines(Paths.get("data.txt"));
```

The `Files` class exposes a static method `lines(path)` that creates a stream to a file that will process it line-by-line. We must pass the method a relative path to the file in question represented by a `Path` object. We create such a `Path` through the static `get` method of the `Paths` class. While the first method of creating a `Stream` is more familar, it also partially defeats the purpose of using a `Stream` in the first place. By creating a `List` first, we must read the entire contents of the file into memory upfront whereas if we create the `Stream` directly, we do not incur this upfront cost.

9.2.2 Stream Processing with Higher-order Functions

With a stream in hand, we do not directly traverse it to perform transformations and analyses over the data. We instead build up a *pipeline* of operations to perform to the stream. These operations are broken up into three sorts corresponding to three methods of the `Stream` class:

- `Stream<U> map(Function<T, U> f)`: transforms the elements of the stream by applying the function `f` to each element.
- `Stream<T> filter(Function<T, Boolean> f)`: filters the stream, keeping all the elements of the stream that produce `true` when given to the function `f`
- `U reduce(U init, BiFunction<U, T, U> f)`: also called *fold*, reduces the stream to a single value by starting with the given initial value and then applying `f` to each element, accumulating a final result in the proces.

As an initial example, we can count the number of occurrences of strings in our stream the start with an "h", ignoring case with:

```
stream.map(s -> s.toLowerCase())
  .filter(s -> s.startsWith("h"))
  .count()
```

The call to `map` makes all the strings lower case. The call to `filter`, removes all strings that do not start with "h". Note that the `map` occurs first in our method chain, guaranteeing that all the strings are lower case. Finally, we then count the number of strings that are left using the `count` method which is a special case of `reduce`.

Anonymous Functions

The arguments to `map`, `filter`, and `reduce` are all functions. As a result, we call these three methods *higher-order functions* because they are functions that take other functions as arguments. In Java 8, we create these function-arguments using *anonymous function values*, also called *lambdas*.

For example, in the following call to `map` from above:

```
stream.map(s -> s.toLowerCase())
```

The single argument to `map` is the lambda `s -> s.toLowerCase()`. The lambda is a function consisting of a single argument called `s`. The body of the lambda is an expression that is evaluated and the resulting value returned when the lambda is called; here, the function produces the lower case version of its argument through the `String` class's `toLowerCase()` method. This lambda behaves similarly to this named static method declaration `f`:

```java
public static String f(String s) {
    return s.toLowerCase();
}
```

As a convenience, Java will also *infer* the types of the arguments and the return type of the lambda so that you don't have to write them down.

Stream Operations

Surprisingly, we can decompose any operation we'd like to perform over a sequential structure as a combination of maps, filters, and folds. By doing so, we gain a concise, readable description of what our operation is doing. As a case in point, consider the example from above that counts the number of words that start with a "h" in a case-insensitive manner:

```java
int count = stream.map(s -> s.toLowerCase())
  .filter(s -> s.startsWith("h"))
  .count();
```

versus a more traditional approach with loops:

```java
int count = 0;
for (String s : list) {
    if (s.toLowerCase().startsWith("h")) {
        count += 1;
    }
}
```

It is immediately clear what the first code snippet is doing whereas we must derive that same meaning by reasoning about the various control structures in the second code snippet.

However, decomposing problems using map, filter, and reduce is not natural at first, especially when we have been trained to program in an imperative style in Java. Let's take a closer look at these three fundamental operations over sequences to get a better feel for how to use them on real-world problems.

Map A mapping operation takes a sequence and transforms it into a new sequence by applying a function to every element of that list. For example, this call to `map` transforms a stream of strings into the same stream but with all of the strings in lower case:

```java
stream.map(s -> s.toLowerCase())
```

The mapping operation does not necessarily need to preserve the carrier type of the stream. For example, this call to `map` transforms a stream of strings into a stream of the lengths of strings:

```java
stream.map(s -> s.length())
```

We also can chain together `map` calls together to perform a series of transformations. This series of `map` calls produces a stream of booleans that indicate if the string has odd length:

```java
stream.map(s -> s.length()).map(n -> n % 2 == 1)
```

Note that the `map` method returns the same `Stream` object, so we can *chain together* calls to `map` to apply them in sequence.

When transforming data with `map`, we sometimes frequently wish to map the elements of a list to multiple datum. We can do so by utilizing a tuple, a fixed-size heterogeneous sequence. More modern programming provide built-in facilities for creating and extracting data from tuples. In Java, we have to create a class for a tuple, one for each size we wish to support. For example, here is a generic pair class (a tuple of two elements):

```
public class Pair<T1, T2> {
    private T1 fst;
    private T2 snd;
    public Pair(T1 fst, T2 snd) {
        this.fst = fst;
        this.snd = snd;
    }
    public T1 getFst() { return fst; }
    public T2 getSnd() { return snd; }
}
```

With this, we can map data onto pairs. For example, perhaps we want to map strings onto whether they are odd length while also preserving the strings themselves. We can do this with the following calls to `map`:

```
  stream.map(s -> new Pair<>(s, s.length()))
        .map(p -> new Pair<>(p.getFst(), p.getSnd() % 2 == 1))
```

The result of this expression is a `Stream<Pair<String, Boolean>>` where each pair is a string and a boolean indicating if the length of the string is odd.

Filter A distinguishing feature of `map` is that it preserves the structure of the sequence. That is, it does not delete or add elements to the sequence; it simply transforms the elements of the sequence. To remove elements from a stream, we appeal to another operation, `filter`. Like `map`, `filter` takes a higher-order function as an argument. However, unlike `map` which takes a function from the stream's carrier type to some other type, `filter` takes a function from the stream's carrier type to `boolean`.

For example, if we have a stream of integers (a `Stream<Integer>`), we can keep all integers that are non-negative with the following `filter` expression:

```
stream.filter(n -> n >= 0)
```

Note that the function we provide to `filter` tells us which elements to *keep* in the stream. In other words, if the function returns `true` for an element, `filter` keeps the element; if the function returns `false` for an element, `filter` removes it.

Reduce Another distinguishing feature of `map` as well as `filter` is that it operates on each element of the stream *independently*. That is, the functions passed to `map` and `filter` cannot make decisions based on anything other than individual elements of the stream. However, we would like to be able to perform operations that "summarize" the stream by accumulating a value built up by analyzing each element of the stream. The `reduce` method allows us to do this by allowing us to supply a function that takes two values:

- The value accumulated so far and
- The current element of the stream

and returns an updated accumulated value. For example, we can sum up a stream of integers using the following call to `reduce`:

```
stream.reduce(0, (acc, n) -> acc + n)
```

In addition to the function value, `reduce` also takes an *initial value* for the accumulated value. In the event that the stream is empty, the initial value is returned unchanged. If we wish to sum up the number of elements in this stream, we can change the call to `reduce` as follows:

```
stream.reduce(0, (acc, n) -> acc + 1)
```

Unlike `map` and `filter`, `reduce` is a *consuming* operation over the stream. Because it necessarily traverses the elements of the stream, it is not lazily evaluated like `map` and `filter`. Furthermore, unlike `map` and `filter`, applying a consuming operation over a stream "uses it up" so that no further operations can be applied to that particular stream.

Finally, the presented `reduce` method is the simple version where the type of the accumulated value is the same as the carrier type of the stream. To accumulate a value of a type different from the `Stream`, you must use the following three argument version of `reduce`:

```
  reduce( /* initial accumulated value */
        , /* binary function: accumulated value, element -> new accumulated value */
        , /* binary function: two accumulated values -> new accumulated value */ )
```

The third argument, the *combiner function*, allows `reduce` to operate over different parts of the stream and then combine sub-accumulated results into a final solution[3] As a convenience, the `Stream` class also provides two additional methods that specialize the behavior of `reduce` to some common cases:

- `count()`: returns the number of elements in the stream.
- `forEach(action)`: applies the `action` function—which takes an element of the list and returns no value—to each element of the stream.

9.2.3 Functional Interfaces*

If a lambda is indeed a value, then it must have a type. What is the type of a lambda? In Java, the answer is (unfortunately) much more complicated than you might expect.

Before lambda expressions and Java 8, developers specified higher-order functions through classes. For example, the `Thread` class found in the `java.lang` package represents a thread of execution in our program. We can use threads to specify multiple threads of executions to run concurrently—*i.e.*, at the same time—in our program. The primary `Thread` constructor takes a `Runnable` object as input. `Runnable` is an interface with the following simple definition:

```
public interface Runnable {
    public void run();
}
```

[3]This implies that the binary function is an associative operation because the order that the elements are traversed is unspecified.

The single method `run()` is the action that the created `Thread` will execute.

To create a `Thread`, we must therefore create a class that implements `Runnable` to supply to our thread. Before Java 8, we had two mechanisms for doing this: create a new class whose purpose is to be supplied to our `Thread` class or use an *anonymous inner class* which allows us to define an object that implements an interface without naming a class, much like how an lambda defines an unnamed function.

As an example, here is how we might use an anonymous inner class to create a `Thread` without having to create a new class:

```
Thread t = new Thread(new Runnable() {
    public void run() {
        System.out.println("Hello World!");
    }
});
```

This thread, when run, will print “Hello World!” to the console. The syntax of an anonymous class is:

```
new <interface name>() {
    // interface implementation goes here
}
```

The result of this anonymous inner class expression is a new object that is a subtype of the given interface.

While anonymous inner classes give us significant flexibility in how we declare objects, it is overkill for situations like this where our object only needs to provide *stateless behavior*. The primary reason lambda expressions were introduced into Java 8 was to give more concise syntax for these situations. Indeed, we can use a lambda expression instead of an anonymous inner class instead in this case:

```
Thread t = new Thread(() -> {
    System.out.println("Hello World!");
});
```

By enclosing the body of the lambda in curly braces, we can specify statements like normal for its body.

For backwards compatability’s sake, the designers of Java needed to give the lambda expression a type such that we can use a lambda expression in place of a `Runnable` while still also allowing old code that used anonymous inner classes that implement `Runnable` to work. Indeed, we can assign our lambda expression to a variable of type `Runnable` and it works:

```
Runnable lam = () -> {
    System.out.println("Hello World!");
};
```

However, we can also assign the lambda expression to an interface type `Action` defined as follows:

```
@FunctionalInterface
public interface Action {
    public void action();
}
// ...
Action act = () -> {
    System.out.println("Hello World!");
};
```

How does this work? Java defines a notion of a *functional interface* which is an interface that:

- Is marked with the `@FunctionalInterface` annotation in its declaration.
- Exposes a single method called its *functional method*.

The functional method defines the type signature of a particular lambda expression, *e.g.*, the `Action` functional interface above defines a type signature for lambdas that take no arguments and return no values. Any lambda that has the same type signature as a functional interface's method is considered a subtype of that functional interface. This allows the same lambda to be both assignable to a `Runnable` as well as an `Action`.

Thus, for any lambda, we can cook up an appropriate functional interface to describe its type. However, to save us that work, the `java.util.function` package defines a number of generic functional interfaces that cover most of the cases we're interested in. For example:

- `Function<T, R>` is the type of lambdas that take a `T` as input and produce an `R` as output.
- `BiFunction<T, U, R>` is the type of lambdas that take a `T` and `U` as input and produce an `R` as output.
- `Predicate<T>` is the type of lambdas that take a `T` as input and produce a boolean as output.
- `Supplier<T>` is the type of lambdas that take no arguments and produce a `T` as output.

The `Stream` class as well as other standard library methods that use higher-order functions use these pre-defined functional interfaces.

Chapter 10

Hierarchical Structures

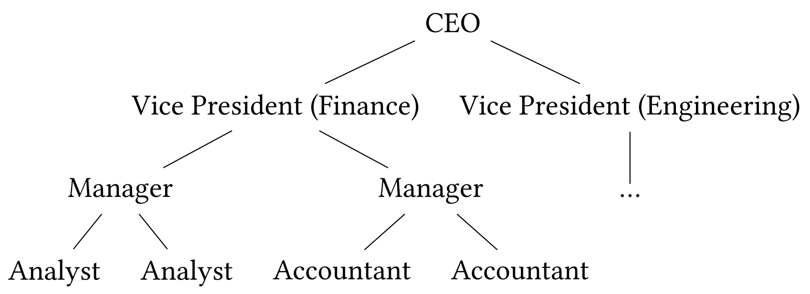
So far, we have studied sequential structures that assign each of their elements an index in the structure. This imposes an ordering relationship between elements based on this index. Now we will examine more complex relationships that we can impose between elements in a structure. The first of these is the *hierarchy* which establishes *child-parent* relationships between values.

10.1 Hierarchical Structures

Hierarchies occur naturally in all sorts of data. For example, consider the following domains:

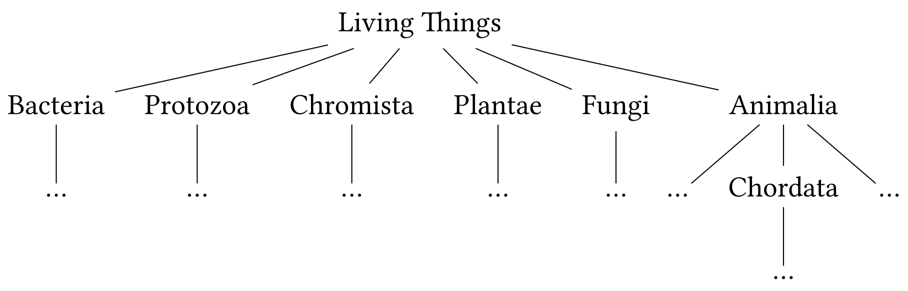
1. The reporting structure in a large corporation.
2. Classification of animals.
3. HTML documents, *i.e.*, web pages.

All of these domains possess a hierarchical structure. A corporation's reporting structure may look like this:



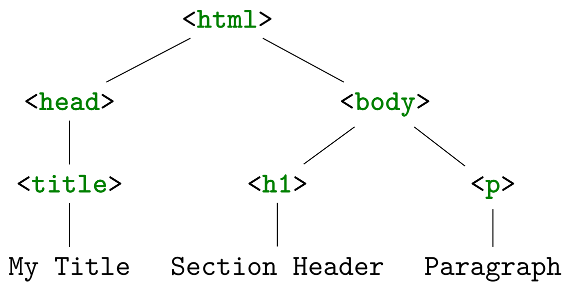
The various individual contributors to a company (*e.g.*, analysts, accountants, and programmers) report to their managers. The various managers within a division report to the vice president of the division. Finally, the various vice presidents report to the CEO who sits at the top of the reporting structure.

Living organisms are classified as follows:



Biologists divide up living things into a hierarchy of classes: kingdoms, phylums within those kingdoms, classes within those phylums, and so forth.

Finally, HTML documents have a hierarchical structure as well. Most web browsers allow you to view the source of a webpage in a tree-like format, *e.g.*, in Chrome: Options → More Tools → Development Tools. Every HTML document begins with an outer `<html> ... </html>` tag. Inside this tag are the elements of the webapge. For example `<head> ... </head>` contains header information about the page and `<body> ... </body>` contains the actual page content. Here is a barebones HTML document with the hierarchy of tags made explicit:



Note that the structure imposed by the hierarchies in all three examples is essential. For example, imagine if we wrote down the corportation reporting structure down as a list:

> CEO, Vice President (Finance), Manager, Analyst, Analyst, Manager, Account, Account, Vice President (Engineering).

From the list, it isn't clear who reports to who in the company! Our list representation of the company *loses* the reporting structure that the hierarchy captured.

10.2 The Tree Abstract Data Type

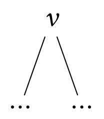
From our discussion above, it is clear that a list is insufficient for representing data that possess hierarchical relationships. We instead use a different abstract data type, *the tree*, to capture these relationships.

A tree is a data type that encodes a hierarchical structure among its elements. The meaning of these relationships is domain-specific. As a result, trees typically don't have a fixed interface like a list—the operations we'd like to perform depend heavily on what the tree is used for. Nevertheless, we'll study some essential core operations and programming techniques over trees that we can adapt to a variety of situations.

To begin, we'll examine the simplest form of a tree to understand these basics. Trees have an elegant recursive definition. A tree is either:

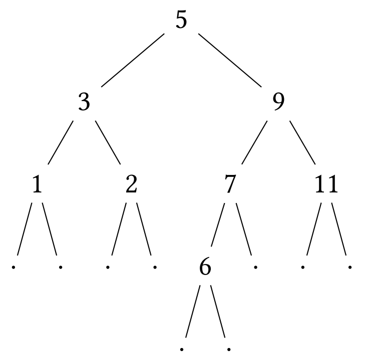
- An *empty leaf* or
- A *node* consisting of an element, a left sub-tree, and a right sub-tree.

Such a tree is called a *binary tree* because its nodes feature two *children*, the sub-trees rooted at that node. We can visualize the second case below:



We typically denote the left and right sub-trees, `left` and `right`, respectively. Note that these sub-trees are simply recursive occurrences of our tree definition—they'll either be empty or be a tree itself.

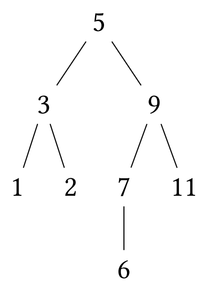
As a concrete example, consider the following tree of integers:



The leaves of the tree are denoted by single dots (·). The top-most element of the tree is called its *root*—here the element 5 is the root. The root has two sub-trees. The left sub-tree contains the elements 1, 2, and 3. The right sub-tree contains the elements, 6, 7, 9, and 11. We can identify any of the sub-trees by its root, *e.g.*, the sub-tree *rooted at* 3 contains itself, 1, and 2. The sub-tree rooted at 7 contains itself and 6. As a degenerate case, the sub-tree rooted at 11 only contains itself, but it is still a tree, nevertheless.

For any two elements in the tree we can talk about the relationship induced by the tree's structure. For example, 7 appears as the root of the left sub-tree rooted at 9. Therefore, we say that 7 is a *child* of 9; conversely, 9 is a *parent* of 7. We'll use all sorts of similar terminology to denote these parent-child relations as is appropriate for the domain, *e.g.*, subordinate and boss for the corporate domain or subclass and superclass for the living organism domain.

Drawing out the empty leaves is usually unnecessary. Therefore, we frequently leave them out to simplify the diagram:



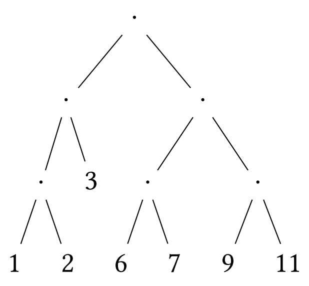
With this diagram, the elements 1, 2, 6, and 7 are the *leaves* of the tree since they have no children. The remaining elements form the *interior nodes* of the tree, nodes with at least one non-empty child.

10.2.1 Placement of Data

Our initial definition of a tree places the data at the *interior nodes* of the tree—*i.e.*, the non-leaf nodes of the tree. However, there is nothing essential about this choice. Indeed, we may give an alternative definition of a tree that places the data at the leaves rather than the nodes: A tree is either:

- A *leaf* containing a value or
- A *node* containing a left and right sub-tree.

With this definition, our sample tree above may be represented as follows:



While this tree is structurally distinct from our original tree, they both encode the same information. Choosing one representation over another is simply a matter of choosing the representation that best fits the domain the tree is being used in.

10.3 Tree Representation and Operations

In essence, a tree is a linked list except that instead of a single `next` field that contains the “rest of the list”, it has two fields `left` and `right` that contain the “rest of the tree”. Therefore, we adopt a similar strategy to represent a tree in Java—a class to represent the nodes of a tree and a class to represent the tree itself:

```
// In Node.java

// In Tree.java
public class Tree<T> {
    private static class Node<T> {
        public T value;
        public Node<T> left;
        public Node<T> right;
        public Node(T value, Node<T> left, Node<T> right) {
            this.value = value;
            this.left = left;
            this.right = right;
        }
        public Node(T value) { this(value, null, null); }
    }

    private Node<T> root;
    public Tree() { root = null; }
}
```

Here, the leaves of the tree are represented with a `null` `Node<T>` value.

Because our tree contains data, we ought to perform similar sorts of operations that we can perform over lists, *e.g.*, adding elements, querying for an element, or checking the size of the container. Imagine implementing this last operation, `size()` for a linked list. We would maintain a `cur` reference to the current node in the list we are examining and increment a counter for each one. Let's try the same approach for our tree:

```
// In the Tree class...
public int size() {
    Node<T> cur = root;
    int size = 0;
    while (cur != null) {
        size += 1;
        cur = cur.left;
        // But what about cur.right...?
    }
    return size;
}
```

However, we run into a problem if we try to apply our linked list traversal techniques to trees. This attempt at `size()` traverses the left-hand nodes of the tree but doesn't visit the right-hand nodes. But once we leave a node, we have no way of coming back to it to visit those nodes!

To remember these nodes, we need to appeal to an auxiliary data structure, *e.g.*, a list.

```
// In the Tree class...
public int size() {
    List<T> pending = new LinkedList<>();
    pending.add(root);
    int size = 0;
    // Loop invariant:
    //   pending contains the current frontier of nodes
    //   that we still need to visit.
    while (pending.size() != null) {
        Node<T> cur = pending.remove(0);
        if (cur.left != null) { pending.add(cur.left); }
        if (cur.right != null) { pending.add(cur.right); }
        size += 1;
    }
    return size;
}
```

Here, we use a list essentially like a *queue*, adding nodes to be explored to the end of the list and then removing nodes to visit from the front.

This approach works, however, the use of an auxiliary data structure is undesirable. Furthermore, our solution does not reflect the recursive definition of the tree. Because of this, we'll pursue a *recursive* definition of the `size()` operation for trees that mirrors our definition for a tree.

In the absence of a particular programming language, we can define the size operation as follows:

- The size of a *leaf* is 0.
- The size of a *node* is one plus the sum of the sizes of its left and right subtrees.

There are several ways to reflect this definition in Java with varying trade-offs of complexity, elegance, and handling of corner cases. Here, we present a particular style that allows our code to reflect this definition directly:

```java
// In the Tree class...
private static int sizeH(Node<T> cur) {
    if (cur == null) {
        return 0;
    } else {
        return 1 + sizeH(cur.left) + sizeH(cur.right);
    }
}

public int size() { return sizeH(root); }
```

We establish a static helper function, `sizeH`, that computes the size of a tree rooted at a given `Node<T>`. The method proceeds by case analysis on the shape of that `Node<T>`—it is either a leaf (`null`) or a node (non-`null`). In this manner, the helper function mirrors the pseudocode definition given above. Finally, we define the actual `size` method to simply call this helper method starting with the `root` of the tree.

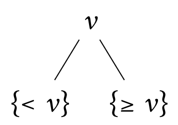
10.4 Binary Search Trees

Before we discuss other tree operations, we must narrow our domain of interest to take advantage of the hierarchical relationships that the tree offers. Recall that linear search over an unsorted sequential structure has $\mathcal{O}(n)$ time complexity. However, if the structure is already sorted then we can employ binary search which has $\mathcal{O}(\log n)$ time complexity. The catch is that we must now keep the structure sorted which requires additional work on top of the sequential operations we've discussed previously.

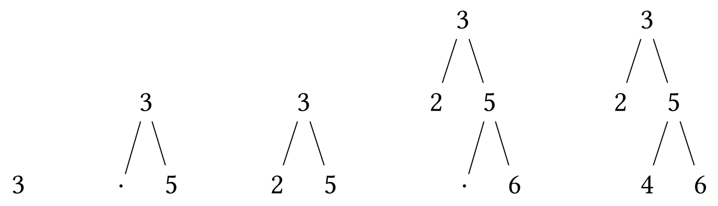
A *binary search tree* is a tree-based structure that maintains the sortedness property among its elements. It does this by way of an invariant that is baked into the definition of the tree. A binary search tree is a tree consisting of either:

- An empty leaf.
- A node consisting of a value and left- and right-subtrees with the property that all the elements in the left subtree are *less than* the value and all the elements in the right subtree are *greater than or equal* to the value.

We can visualize this *binary search tree invariant* as follows:



This invariant gives us guidance as to where to place elements in the tree. For example, consider starting out with a empty binary tree and then adding the elements 3, 5, 2, 6, and 4. Here is the evolution of our tree after each insertion.



In general, our insertion strategy is to traverse the tree according to the binary search tree invariant to find a leaf. We then replace the leaf with a node containing the value to be inserted. In the above example:

1. Initially, we replace the single leaf of the empty tree with a node containing the value 3.
2. To insert 5, we note that 5 is greater than 3, so we recursively dive into the right-hand subtree, find that it is a leaf, and replace it with a node containing 5.
3. To insert 2, we note that 2 is less than 5, so we recursively dive into the left-hand subtree, find that it is a leaf, and replace it with a node containing 2.
4. To insert 6, we note that 6 is greater than 3 and 5, so it goes into the right-most subtree.
5. Finally, to insert 4, we note than 4 is greater than 3 but less than 5, so goes into the left subtree of 5.

We can generalize these examples into a procedure for inserting elements into a binary search tree. When inserting a value v into a binary search tree:

- If you are inserting into a leaf, then replace that leaf with a node containing v and no left or right subtrees.
- If you are inserting into a node that contains some value v', then recursively insert into the left subtree if $v < v'$, otherwise recursively insert into the right subtree.

We may realize this in Java as follows:

```java
public class BinarySearchTree<T extends Comparable<T>> {
    // Node class same as before...
    private Node<T> root;
    public BinarySearchTree() { root = null; }

    /** @return the updated tree after inserting h into the given tree */
    private static Node<T> insertH(T v, Node<T> cur) {
        if (cur == null) {
            return new Node<>(v);
        } else {
            if (v.compareTo(cur.value) < 0) {
                cur.left = insertH(v, cur.left);
            } else {
                cur.right = insertH(v, cur.right);
            }
            return cur;
        }
    }
    public void insert(T v) { root = insertH(v, root); }
}
```

The definition of the `BinarySearchTree<T>` class is identical to our regular `Tree` class. The exception is that in order to maintain the binary search tree invariant, we must be able to compare elements contained within the `Tree`. This means that we must constraint the generic type `T` to be any type that implements the `Comparable<T>` interface, *i.e.*, `T` defines how to compare elements against itself.

The definition of `insert` follows the skeleton we established for `size` above. However, unlike `size`, `insert` modifies the underlying tree once it finds a leaf—a node that is `null`. To avoid having to write `null` checks for each of the sub-trees of a node, we employ a recursive design pattern called the *update pattern*. Our recursive method, `insertH` takes the `Node<T>` that is the root of the tree as well as the element to insert as input. The method also returns a value—*the updated root*—as output. In the case where we insert into a leaf, the root is `null`, so the method returns a new node. In the case where we insert into a node, the root is non-`null`, so the method simply returns the node that was passed to it. However, along the way, `insertH` modifies this node with an updated sub-tree.

We can think of `insertH` as returning an *updated* version of its input `Node<T>`. This is why the `public` version of `insert` has the following form:

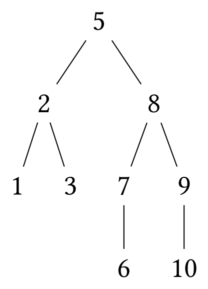
```
root = insertH(v, root);
```

We have updated the `root` of the tree with the result of inserting `v` into the tree.

10.4.1 Tree Traversals

Next let's revisit traversal of a tree. `size()` is a simple example of a tree traversal method. However, the *order* we visit the elements of the tree is irrelevant in calculating the size of the tree. In contrast, imagine a method `toString` that prints the elements of the tree. Here, the order in which we visit the elements does matter.

Consider the following sample tree:



And the following pseudocode description of `toString`:

- If the tree is a leaf, its string representation is the empty string.

- If the tree is a node, its string representation is the string representation of the value, followed by the string representations of the left-hand and right-hand trees, in-order.

This version of `toString` first "stringifies" the value at a node before recursively descending into its sub-trees. This results in the following output for the sample tree:

```
[5, 2, 1, 3, 8, 7, 6, 9, 10]
```

This is an example of a *pre-order traversal* of the tree where we "visit" the value at the node first, then the left-hand sub-tree, and the right-hand sub-tree.

We can exchange this order to obtain two other traversal strategies:

- *In-order traversal*: Recursively process the left-hand sub-tree, "visit" the value at the node, recursively process the right-hand sub-tree.
- *Post-order traversal*: Recursively process the left-hand sub-tree, recursively process the right-hand sub-tree, and "visit" the value at the node.

An in-order traversal of the sample tree yields the list `[1, 2, 3, 5, 6, 7, 8, 9, 10]`. The post-order traversal of the sample tree yields the list `[1, 3, 2, 6, 7, 10, 9, 8, 5]`.

Each traversal order has its use cases. In particular, an in-order traversal of a binary search tree yields the elements of the tree in sorted order. Pre-order traversal provides a convenient way for *serializing* a tree into a linear form appropriate for storage in a file that can be used to recreate the tree later. If we interpret the interior nodes of the tree as operators and the leaves as values, post-order traversal yields *postfix notation* or *reverse polish notation (RPN)* which does not require expressions to be parenthesized. For example, the mathematical expression written in traditional infix style $3 \times (4 + 5)$ has the unambiguous representation in RPN: $3\ 4\ 5\ +\ \times$.

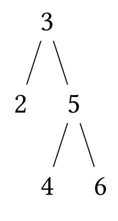
10.4.2 Complexity Analysis

Finally, let's consider the complexity of the various tree operations we've discussed in this chapter.

Time Complexity

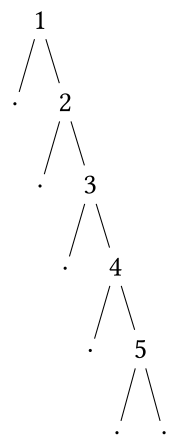
The various traversals, like their sequential counterparts, visit every element of the structure; they, therefore, all take $\mathcal{O}(N)$ time where N is the number of elements in the tree.

More interesting is the cost of lookup and insertion into a binary search tree. In the worst case of lookup, we search one path from the root of the tree to one of its leaves. For example, in the following binary search tree:



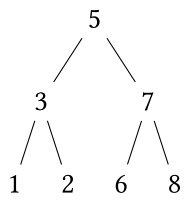
If we look for the value 4, we'll visit the nodes 3, 5, and 4 during the search process. Thus, the runtime of lookup is dependent on the length of such a path.

Let's consider a degenerate example of a binary search tree.



This tree is a binary search tree, however, it is far from an ideal one. It is essentially a linked list! Searching this binary search tree takes $\mathcal{O}(N)$ time in the worst case, the same as linked list search.

Now let's consider an ideal binary search tree:



This binary search tree has three *levels* of nodes. The first level contains the element 5. The second level contains the elements 3 and 7. The third level contains the elements 1, 2, 6, and 8. Each of these levels are *full*, that is, they have the maximum number of possible elements. We call such a tree *perfect*—all interior nodes have two children and all leaves exist at the same level.

To assess the length of a path in this perfect tree from root to leaf, we must consider the number of nodes at each level of a perfect binary tree. The first level contains 1 element, the second contains 2, the third contains 4, the fourth level contains 8, the fifth level contains 16, and so forth. It is reasonable to hypothesize that the number of nodes at level i is 2^i. This turns out to be true and provable with a quick proof by mathematical induction:

Claim 1. *The number of nodes at level i of a perfect binary search tree is* 2^i.

Proof. Proof by induction on the level i.

- $i = 0$: at level 0 (the first level), there is 1 node, the root, and $2^0 = 1$.
- $i = k + 1$: by our inductive hypothesis, level k contains 2^k nodes. Because each node of level k contributes two nodes, a left and right child, to level $k + 1$, then the number of nodes at level $k + 1$ is $(2^k) \cdot 2 = 2^{k+1}$. □

The total number of nodes in a perfect binary search tree of height h is therefore given by summing up the nodes at each level:

$$N = \sum_{i=0}^{h} 2^i = 2^0 + 2^1 + 2^2 \cdots + 2^h$$

This sum relates the total number of nodes of the tree with its height. It has a closed form solution:

$$N = \sum_{i=0}^{h} = 1 - \frac{2^{h+1}}{2-1} = -(1-2^h) = 2^{h+1} - 1.$$

A perfect binary tree of height h has $2^{h+1} - 1$ nodes. From this, we can solve for h in terms of N.

$$\begin{aligned} N &= 2^{h+1} - 1 \\ N + 1 &= 2^{h+1} \\ \log_2 N + 1 &= \log_2 2^{h+1} \\ \log_2 N + 1 &= h + 1 \\ \log_2 N + 1 - 1 &= h \end{aligned}$$

Thus, the height is bounded by $\log N$. When the tree is perfect, lookup has worst-case time complexity $\mathcal{O}(\log N)$. Note that insertion into a binary search tree operates identically, so it too has worst-case $\mathcal{O}(\log N)$ time complexity in this situation.

However, what is the appropriate average case time complexity of lookup? This turns out to be difficult to analyze precisely—what is the layout of the average tree? This depends on the effects of the insertion and deletion operations performed on the tree. In particular, deletion favors the rotation of one side of the nodes, so we might expect that the tree becomes more unbalanced with repeated deletions.

To make progress, we can restrict our question to layout of the average tree created by only a chain of insertion. It turns out that, on average, the height of such a tree is $\mathcal{O}(\log N)$, *i.e.*, the average height is within some constant factor of the optimal height. Therefore, in this situation, the average case of insertion is $\mathcal{O}(\log N)$. However, even when we consider only insertions, we can still obtain the degenerate binary search tree if we insert elements in-order.

In general, our current insertion policy does not allow us to maintain a *balanced* tree shape, one that looks roughly like a perfect tree. To get around this problem, we employ various *balancing techniques* to maintain a balanced tree while maintaining good performance of our fundamental tree operations. Examples of trees employing balancing techniques include AVL trees, red-black trees, and B-trees. All of these structures place additional constraints or invariants on the structure of the tree that ensure that it remains well-balanced.

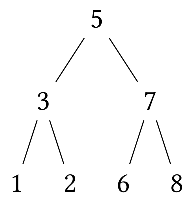
Space Complexity

None of the operations we've examined requires additional heap allocations. However, we must be cognizant of the fact that our recursion itself takes up space—namely space on the stack. With non-recursive functions, we always make a constant number of function calls relative to the input. However, with recursive functions, the number of function calls depends on the size of the input in some way. This means potentially that we require a non-constant amount of stack space to execute our functions!

Recall that the pending function calls in our program occupy one *stack frame* per call. What is the largest number of pending function calls that we build up while executing our operations over trees? First let's consider the `size` operation. Recall we implement `size` using the following recursive helper function:

```java
private static int sizeH(Node<T> cur) {
    if (cur == null) {
        return 0;
    } else {
        return 1 + sizeH(cur.left) + sizeH(cur.right);
    }
}
```

First, note that we will make one recursive call per element of the tree. However, it is not necessarily the case that all of these function calls will be active at the same time. To see this, consider the binary search tree from before:



and consider the implementation of `sizeH`. Keeping in mind that we evaluate expressions left-to-right, `sizeH` descends down the left-hand side of the tree (exploring `cur.left`) and then the right-hand side of the tree (exploring `cur.right`). Before we explore the right-hand side of the tree, we return from all the recursive calls corresponding to exploring from the left-hand side.

Dually, any given call to `sizeH` does not return until its recursive calls to its left and right sub-trees return. That means that the number of recursive calls active at a given node corresponds to the *length of the path* from the root to that node. For example, in the above tree, when we make the recursive call to compute the size of the tree rooted at 6, the recursive calls to 7 and 5 are pending. Thus, the amount of stack space used up by our recursion is proportional to the length of the longest path from root to a leave of the tree.

The length of such a path depends on the shape of the tree as discussed in the previous section. With a degenerate tree, the maximal length is $\mathcal{O}(N)$; with a balanced tree, the maximal length is $\mathcal{O}(\log N)$. Thus, we expect that the space complexity of our tree operations is $\mathcal{O}(\log N)$, assuming that the tree is roughly balanced. Note that this space complexity cost is essential to the operations; it is not an artifact of our implementation choice of recursion. In particular, if we used iteration to implement `sizeH`, we would need an explicit stack data structure to hold pending nodes that we must visit. We can perform a similar analysis to discover that we will need to hold at most $\mathcal{O}(\log N)$ nodes in our stack at any given point in time, assuming that the tree is roughly balanced.

Chapter 11

Mapping Structures

Some data exhibit a sequential relationship between elements. Other data exhibit a hierarchical relationship between elements. And yet, some data exhibit a *mapping relationship* between elements. For example:

1. A language dictionary maps words to definitions.
2. A bank account database maps account numbers to balances.
3. The Dewey Decimal Classification system maps Dewey Decimals to books.
4. In an American grade school, students learn how to map states to their capitals.
5. An array maps indices to values.

For these data, we have *mapping data structures* that allow us to efficiently query these elements based on their relationships to each other.

11.1 An Example: Language Dictionaries

To get an intuitive feel for the sorts of operations we might ask of our mapping types, let's consider a concrete example: *language dictionaries.* What might we do with such a dictionary? Initially, we would start with an *empty* dictionary, so it would be prudent to put entries into it. For example, we might want to *add* the definition for cat: "a small domesticated carnivorous mammal with soft fur, a short snout, and retractile claws". If our definition changes as language is an amorphous beast, we might fix it up by putting a new, *updated* definition of "cat". We should also be able to *remove* the definition of "cat" if the powers-that-be decide that cats are no longer a thing. Finally, after adding a number of definitions, it makes sense to ask for the *size* of the dictionary which is the number of entries it contains.

The most important operation we can perform on a dictionary is *lookup* a word for its corresponding definition—alternatively get the definition for a given word. When doing so, we'd like to avoid searching the whole dictionary, *e.g.*, by organizing the dictionary in lexicographical order of the words, we can quickly find where "cat" sits in the dictionary. Our dictionary structure ought to support fast lookup of these words. Note that if the word is not in the dictionary, then we should signal an error that the word was not found. Correspondingly, we would like to be able to check to see if our dictionary *contains* an entry for the word before we try to perform a lookup.

Finally, we can think of our dictionary as a collection of entries where each entry is a pair of a word and its definition. But alternatively, we can think of it as two separate collections of words and definitions. We may want to get this collection of words and collection of definitions separately to analyze them.

11.2 The Map ADT

Each of the operations we'd like to perform on a language dictionary translates directly into a corresponding operation on the map ADT:

Let's turn our intuition of how a language dictionary works into a general abstract data type capturing the essence of a mapping data structure. The map abstract data type captures a mapping from keys of one type `K` to values of another type `V`. We'll call each key-value pair, `(k, v)`, an *entry* in the map. In our concrete example above, our keys are words (strings) and our values are definitions (strings). Note that the type of keys and values can be distinct. For example, an account number might be a string, but the value that an account number maps to, the balance, is an integer.

- `void put(K k, V v)`: put an entry for key `k` (of type `K`) in the map, associating it with value `v` (of type `K`). If an entry for `k` already exists in the map, we overwrite the old value with this value `v`.
- `V remove(K k)`: removes the entry for key `k` from the map if it exists, returning its corresponding value.
- `int size()`: returns the number of entries in the map.
- `boolean containsKey(K k)`: returns `true` if the map contains an entry for key `k`.
- `V get(K k)`: returns the value `v` associated with the key `k`; throws an error if `k` is not present in the map.
- `List<K> keys()`: returns a list of the keys of this map.
- `List<V> values()`: returns a list of the values of this map.

In Java, we might define the following interface to capture these operations:

```
public interface Map<K, V> {
    public void put(K k, V v);
    public V remove(K k);
    public int size();
    public boolean containsKey(K k);
    public V get(K k);
    public List<K> keys();
    public List<V> values();
}
```

The Java standard library defines an interface `java.util.Map` that captures this abstract data type. It contains these operations (with some slight differences in signatures of methods) along with other methods.

11.3 Association Lists

How might we implement this abstract data type? One way to do it is to realize our map as a list of key-value pairs. Assuming that we have a way of representing pairs of data, for example, a `Pair` class:

```
public class Pair<T, U> {
    public T fst;
    public U snd;
    public Pair(T fst, U snd) {
        this.fst = fst;
        this.snd = snd;
    }
}
```

We can implement the map abstract data type by using a list of pairs. Each pair `(k, v)` is an entry in the map where the first component is a key and the second component is its corresponding value.

Let's consider an example of using an association list to get a feel for how we would implement these operations. The empty map is represented by the empty list:

```
[]
```

Let's consider creating a map from strings to integers, where the integer is the string's size. First, let's add `("dog", 5)` to map. This amounts to adding the pair to the list:

```
[("dog", 5)]
```

When we insert new key-value pairs, we simply continue to add pairs to the list in-order:

```
[("dog", 5), ("doghouse", 8), ("cat", 3)]
```

However, in general, if we add a key that already exists, we must first remove the old key-value pair and then re-add it with the new value. For example, consider fixing our entry for "dog" so that it is correct:

```
     [("doghouse" 8), ("cat", 3)]
--> [("doghouse" 8), ("cat", 3), ("dog", 3)]
```

In general, when inserting into an association list, we must see if the key already exists in the map. If so, we delete its corresponding key-value pair, and then we perform the addition like normal.

The remainder of the operations are straightforward to implement with association lists. The size of the map corresponding to the size of the list, *e.g.*, there are currently three entries in the map above. To see if a list contains a given key `k`, we must perform a traversal of the list to see if one of the pairs has key `k` as its first component. Similarly, if we want to get the value for key `k`, we perform a similar traversal, returning the corresponding value. Deletion is like `get` except we remove the key-value pair from the list. Finally, generating a list of only the keys or only the values also requires a traversal of the list, adding either the keys or values to a new list to be returned.

Note that the description so far hasn't taken the sortedness of the keys into account. We might choose to sort the key-value pairs by the key so that we can perform more efficient lookup, *e.g.*, by using binary search. However, maintaining this sortedness on every insertion makes the insertion operation less efficient and complicates our implementation. For now, we'll choose to leave the keys unsorted when talking about the complexity of association lists, leaving sorting to our BST-based implementation that we discuss next.

The complexity of all the map operations using association lists—because the structure is list-based—correspond precisely to the complexity of the structure's underlying list operations. For example, to put a key-value pair into the association list, we must check to see if `k` is already in the list. In the worst case, we must then remove that old key-value pair in the list and then add the new key-value pair to the list. The check takes

$\mathcal{O}(n)$ time (where n is the number of entries in the map) and the removal takes $\mathcal{O}(n)$ time—both are linear traversals of the underlying list. The addition is either $\mathcal{O}(1)$ or $\mathcal{O}(n)$ depending on our underlying list data structure and where we add onto the beginning of the list or its end. Note that it does not matter where the key-value entry goes. Overall, this means that the time complexity of `put` is the sum of complexity of these three operations, or $\mathcal{O}(n)$. The complexity of all the other map operations is also linear, $\mathcal{O}(n)$, because they all require linear traversals of the underlying list.

11.4 Tree Maps

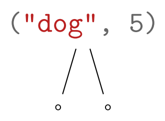
Instead of a list-like structure, we can require that our keys be comparable. By doing this, we can sort our map "by key" to perform more efficient lookup. However, we must maintain the sortedness of our map on every insertion. A binary search tree allows us to do this easily, so we can alternatively implement a map using a binary search tree, typically called a *tree map*.

The elements of a binary search tree are not just single values, but key-value pairs like with association lists. By using a binary search tree, we require that the keys are comparable so that we can use them to determine the placement of key-value pairs into the BST. Using the example above, we can compare strings by considering their lexicographical ordering. For example:

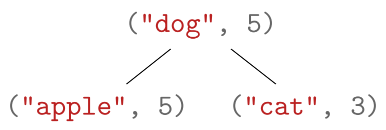
```
"apple" < "banana" -- "a" appears before "b", "a" < "b", in the alphabet
"acorn" < "apple"  -- the 1st letters are the same, but "c" < "p" (2nd letters)
"dog"   < "dogs"   -- the 1st three characters are the same, but "dogs" is longer
                      (the empty letter "comes before" any other letter)
```

We compare both words, letter by letter. If the letters are the same, we move onto the next letter. If they are different, we declare the "smaller" word to be the one that has the "smaller" letter where "smaller" for letters is defined by their underlying character values (which coincides with alphabet order in the case the characters are letters). The `String` class implements the `Comparable` interface so that strings can be compared in this manner.

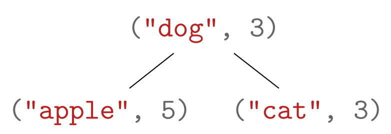
We begin with the empty tree. Inserting the first entry `("dog", 5)` gives us the following tree:



After inserting `("apple", 5)` and `("cat", 3)`, we get the following structure (because `"apple" < "dog"` and `"dog" < "cat"`):



Note that deleting from the BST is undesirable because it will cause the tree to become more unbalanced. So, to update a key's entry, we simply search for that key in the tree and if we find it, we update the corresponding entry:



Otherwise, like the association list, the remaining operations on maps are implemented in terms of the corresponding operations on BSTs. In particular, when we generate the list of keys of the map, if we traverse the BST in-order to generate the list, we receive a sorted list of the keys which may be convenient for a number of applications.

The complexity of the tree map operations comes directly from the complexity of underlying BST operations used to implement them. Critically, unlike the association list which has $\mathcal{O}(n)$ insert and lookup, the BST has $\mathcal{O}(n)$ time complexity (for balanced trees). In practice, the BST is implemented with one of the balanced tree structures discussed briefly earlier—AVL trees, red-black trees, or B-trees. The `java.util.TreeMap` class implements the `Map` interface in Java's standard library and uses a red-black tree structure to maintain balancing.

11.5 Integer Maps

Consider a special case of our map data structure where the keys are integers. For example, our map may be recording an inventory where we are mapping product identification numbers (pids) to the count of such items in the inventory. Let's make the following assumptions:

- The range of valid pids is finite and known, *e.g.*, say we only anticipate at most 1000 products in our stock, so our pids range from 0–999.
- The pids are unique, that is, any two products have distinct pids.

With these two assumptions, we can use an array to implement our map. Our keys, here pids, serve as indices into the array. We say that the ith element of the array contains the count of the item with pid i. Because we know the range of pids is 0–999, it suffices to allocate an array of size 1000 (with indices 0–999) to store these key value pairs.

For example, consider an initially empty inventory. Our backing array would consist of an array of 1000 cells, each containing zero. Note that we obtain this behavior by simply initializing the array as the "zero" initialization value for an int is 0.

```
[0, 0, 0, 0, 0, ...]
```

Next, let's add some values to our inventory. For example, suppose we have five copies of an item with pid 2 in stock. Then we can update the second index in the array:

```
[0, 0, 5, 0, 0, ...]
```

If we want to look up the number of copies of the item with pid 2, we simply look at index 2 in the array. We can update another item, *e.g.*, say the item with pid 1 has 10 copies in stock:

```
[0, 10, 5, 0, 0, ...]
```

If we want to update one of items, *e.g.*, we sold two copies of the item with pid 2, we simply update the corresponding position in the array:

```
[0, 10, 2, 0, 0, ...]
```

With the setup, our key-map operations are very easy to implement:

- `lookup`: a single array read at index `k`—$\mathcal{O}(1)$ time.
- `put(k, v)`: a single array write at index `k` with value `v`—$\mathcal{O}(1)$ time.

Because we rely on array indexing, we gain extremely efficient run times for `lookup` and `put`. However, we encounter a slight technical hiccup with `size`. Technically in the above inventory, even though we haven't explicitly put a count for the item pid 0, it has the default value zero. This is likely correct given our interpretation of the data structure—if we have not updated the map for a particular item, then we do not have any of that item in stock. Thus we could consider the size of the map to be 1000, corresponding to the 1000 pids that we are accurately capturing with this map.

In general, though, this may not be the case. For example, we may map a pid to its manufacturer which we represent as a string. The default value for string (because it is an object), is null, so our initial array looks as follows:

```
[null, null, null, null, ...]
```

Updating a pid, say 2, with its manufacturer updates the backing array as follows:

```
[null, null, "J&J Produce", null, ...]
```

The key thing to note is that the entries for pid = $0, 1, 3, \ldots$, do not contain valid entries. We can get around this by assigning a sentinel value in the range of the map, *i.e.*, setting aside a value to be the "there no value" value. For example, in the above map, `null` could serve as the "no entry" value for a given key. However, we know that this doesn't work in all cases, *e.g.*, if it is possible to use all the values of a given type in the map.

To get around this, we can use the Java 8 `Optional` type which encodes whether we have a value of type `T`. You can think of an `Optional` as a 1-element cell that is either empty or non-empty. For example, our array above would contain:

```
[Empty, Empty, Optional("J&J Produce"), Empty, ...]
```

Where `Empty` corresponds to an empty optional value (its `isPresent()` method returns false), and `Optional(v)` corresponds to an optional value that contains some underlying value (`isPresent()` returns `true` and `get()` produces that value).

11.6 The Set ADT

A mathematical set is a collection of elements without duplicates. The set abstract data type is the realization of this concept in a program—it, too, is a collection of elements without duplicates. For example, we may start with an empty set of numbers:

$$\{\}$$

We can add elements to this set:

$$\{1, 4, 2, 9, 8\}$$

Trying to add a duplicate, *e.g.*, 4 to this set does not change the set because 4 is already in it.

A set acts similarly to a list in that we can add and remove elements as well as iterate over its contents. Unlike a list, a set is not a sequence so the elements do not have indices that we can refer to them by. Importantly, a set allows us to query for elements more efficiently than with a list—we ought to be able to check to see if an element is contained by the set in better-than-$\mathcal{O}(n)$ time.

In addition, we'd like to perform similar operations over our set data type that we would perform over a mathematical set. In particular, we ought to be able to take the union of two sets. The union of two sets (written $A \cup B$ in formal mathematical notation) is simply the combination of all the elements from both sets, respecting duplicates:

$$\{1, 4, 2, 9, 8\} \cup \{3, 2, 4, 6\} = \{1, 3, 4, 2, 9, 8, 6\}$$

The intersection of two sets (written $A \cap B$ in formal mathematical notation) is the set of elements found in both sets:

$$\{1, 4, 2, 9, 8\} \cup \{3, 2, 4, 6\} = \{4, 2\}$$

We can summarize these operations as follows:

- `void add(T v)`: adds `v` (of type `T`) to set—does nothing if `v` is already in the set.
- `boolean remove(T v)`: removes `v` from the set, returning `true` if the removal succeeds, and `false` otherwise.
- `boolean contains(T v)`: returns `true` if `v` is in the set and `false` otherwise.
- `int size()`: returns the size of the set.
- `Set<T> union(Set<T> s)`: returns a new set that is the result of taking the union of this set with the other set `s`.
- `Set<T> intersect(Set<T> s)`: returns a new set that is the result of taking the intersection of this set with the other set `s`.

Sets are seemingly unrelated to the mapping structures discussed previously. However, note that the "no-duplicates" rule of a set is precisely the restriction that map places on its keys—a map may only have one entry per key. Therefore, we can implement a set in terms of a map: a set is simply a map where the values don't matter!

To capture this idea that the "values don't matter", in Java, we may define a type `Unit` that does nothing and acts as a placeholder value:

```java
public class Unit {
    private Unit() { }
    public static final Unit value = new Unit();
}
```

Because `Unit` has no fields or behavior, every value of unit is equivalent to itself. So rather than creating many such empty `Unit` values (via `new`), we hide the constructor to `Unit` by marking it `private` and then provide a single `Unit` value to clients of the class via a static field. This pattern of providing a single instance of a class while disallowing others from creating instances of the class is called the *Singleton Pattern*.

With our `Unit` type in place, we can define a set over type `T` to simply be a map from `T` to `Unit`. Adding an element `v` into the set is equivalent to putting the key `(v, Unit.value)` into the underlying map. All of the other basic set operations act similarly to their map counterparts. The complexity of these operations depend on the complexity of the underlying set. In particular, Java provides the `Set` interface and the

`TreeSet` class which is a red-black tree implementation of the interface. Tree sets (really, tree maps) provide $\mathcal{O}(\log n)$ for insertion, removal, and contains checks.

Union and intersection require a little bit more work. We can implement union by repeated insertions. For example, in the below situation:

$$\{1, 4, 2, 9, 8\} \cup \{3, 2, 4, 6\}$$

We can simply create a new, empty set and then insert each of the elements from the left-hand and right-hand sets into this new set. Of the n elements, each insert takes $\mathcal{O}(\log n)$ time (again, assuming that the tree stays balanced) for an overall runtime of $\mathcal{O}(n \log n)$. We can do a similar thing with intersection, but we only insert an element if both sets contain the given element. This approach requires three $\mathcal{O}(\log n)$ operations per element—two contains checks and an insertion—which results in an overall runtime of $\mathcal{O}(n \log n)$.

Chapter 12

Hashing

So far, we have explored a pair of implementations for our map ADT. Association lists gave us $\mathcal{O}(N)$ lookup, and tree maps gave us $\mathcal{O}(\log N)$ lookup. Can we do better than this? Recall that an integer map is a special case of a map where the keys are draw from a finite set of integers. In this situation, we can use an array to efficiently implement the map ADT where keys are indices and values are elements of the array.

In this reading, we'll breaking apart the assumption of the integer map to generalize the structure to handle a wider variety of domains. By doing so, we'll derive a data structure called a *hash map* which is a very efficient implementation of the map ADT. Along the way, we'll also develop a technique called *hashing* which is one of the most important techniques in computer science with applications towards fingerprinting, compression, and cryptography, among others.

12.1 Revisiting Integer Maps

The first assumption we made when using an integer map was that the space of keys was finite and known. Recall that our running example of an integer map is an inventory where we map product IDs (pids) to inventory sizes. For this example, we assumed that our pids were drawn from the range 0–999. What happens if we don't know what values the space of keys ranges over? For example, we may want to support a number of products beyond 1000.

This seems to pose a problem for our integer map strategy. By definition, an array can only hold a fixed number of elements. With a variable number of possible keys, it seems like we cannot use an array to hold them. This seems like a job for an array list which can hold a variable number of elements, but a deeper problem remains. Consider the following map where we only hold an entry for key value 10000 (and we use `null` as our value indicating that this key has no entry in our map):

```
[null, null, null, ..., v]
                        ^
                   index 10000
```

Because we map keys directly onto indices, our array list would need to contain entries for 0–9999 even though only one entry is in the map for key 10000. This is an extreme waste of space—the size of the backing array or array list must be as large as the largest key that we need to support. However, if the space of keys that the map actually needs to store is *sparse*, *i.e.*, very few keys are required from the range, then we end up wasting lots of space. Ideally, we'd like to avoid using a variable-size structure and instead use a fixed-size structure so that our space usage is not proportional to the size of the key space that we might support (which is very large in practice).

How can we support a variable number of keys using a fixed-sized array? One trick we can try is using the modulo operator to fit the keys into the available space of the array. Suppose that we only allocate an array of size 10 to hold the entries of our map. Then we can simply mod our key by 10 to get an index that fits in the array, *e.g.*,

- The values 0, 10, 20, 30, …, all map to index 0.
- The values 1, 11, 21, 31, …, all map to index 1.
- …
- The values 9, 19, 29, 39, …, all map to index 9.

If our map contains entries for keys 232, 11196, 555, and 8, then our backing array would look like this:

```
[null, null, v1, null, null, v2, v3, null, v4, null]
```

Where `v1` is the value associated with key 232, `v2` is the value associated with key 555, `v3` is the value associated with key 11196, and `v4` is the value associated with key 8. Accessing the entry for key `k` means that we simply look up index `k % 10` in the array.

Of course, the issue here is that if also want to store an entry for key 1992, we have a problem because there is an entry in index 2 already for key 232. This is known as a *collision*—two keys want to use the same position in the array. There are two primary methods of resolving collisions: *probing* and *chaining*.

12.1.1 Probing

When a key's *preferred index* has already been taken by another key, we can *probe* the array for another position (presumably empty) to store the key. Probing means searching the array in a systematic manner starting from a key's preferred index to find an open position to store its value. A simple probing strategy is to simply search successive elements in the array until we find an empty spot, a technique called *linear probing*.

To do this, we must augment what our arrays carry as values. Rather than carrying the values of our map directly, each array index will carry both a key and its corresponding value. This is because with our probing strategy, a cell may no longer correspond to a key's preferred index—some other key might have already taken that spot. By carrying the key, we can ensure that we found the appropriate value by comparing the key we found in the array to our target key.

In the example above, consider starting with an array of five elements. For simplicity's sake, we'll just note the key in our diagram, but keep in mind that we are storing an entire key-value pair in each array slot.

```
[null, null, k1, null, null ,k2 ,k3, null, k4, k5]
```

Suppose that we are adding the key `k6` that has a value of 2. The key `k1` is already in that slot, so we search to the right until we find an empty position and place `k6` there.

```
[null, null, k1, k6, null, k2, k3, null, k4, k5]
```

When we go to look up `k6` in this map, we start at index 2 and scan to the right until we find the entry for `k6`. Now, what happens if we add a key `k7` that has id 8? Index 8 in the array is already taken by `k4`, so we search index 9. However that is taken by `k5`. We proceed by *wrapping around* the array and starting our first at the index 0. This index is empty, so we place `k7` there:

```
[k7, null, k1, k6, null, k2, k3, null, k4, k5]
```

In general, during `put` and `lookup`, our linear search *wraps around* the array, terminating when we reach the preferred index of the key in question.

Using a probing scheme, we achieve $\mathcal{O}(1)$ complexity for our fundamental map operations—`put` and `lookup`—as long as there are no collisions with our keys. However, if there are collisions, we'll need to search part of the array to find our key. In the worst case, all keys have the same preferred index, so we'll need to perform a linear scan of the array which takes $\mathcal{O}(n)$ time. Note that discussing the "average case" here is very difficult because it depends on the likelihood of collisions with our keys which is dependent entirely on the specific keys we add into the map.

12.1.2 Chaining

With probing, we maximize the space in our backing array by checking successive indices for open positions. When our map is *sparse*, this is fine because there's plenty of empty spaces between elements for collisions to be placed. However, if the map is *dense*, *i.e.*, contains many elements, then probing will need to traverse significant portions of the array to find the key of interest. Rather than doing this, we can elect to store all of the keys of a particular id at their associated index. To do this, we store a *list* of key-value pairs at each index of the array rather than a single such pair. For example, we may have the following map:

```
[ ][ ][ ][ ][ ]
 |  |  |  |  |
 |  |  |  |  |--> [k7]
 |  |  |  |--> [k5, k6]
 |  |  --> []
 |  --> [k4]
 -->[k1, k2, k3]
```

Keys `k1`, `k2`, and `k3` all have id 0, so they map onto index 0 in the array. Index 0 contains a list that holds these three key-value pairs. In contrast, there are no stored keys with id 2, so index 2 holds an empty list.

This method of collision resolution is called *chaining*, named as such because these lists look like chains hung off of each array index. Alternatively, we can call each index a *bucket* and the effect of chaining is to store all the key-value pairs of a certain id in their corresponding bucket. In essence, these lists function like the association lists we studied earlier except that all the keys contained in a list have the same id.

By doing this, we no longer have to perform a linear scan of the array. Instead, we perform a linear scan of a bucket's association list. A bucket fills up only with colliding keys, so the amount of keys that we need to traverse is proportional to the number of keys that collide for a particular bucket. In the best case, a bucket contains exactly one key-value pair corresponding to the desired key (in the absence of collisions) for $\mathcal{O}(1)$ lookup. In the worst case, everything has the same key, so a single bucket contains all the entries in the map. In this situation, lookup takes $\mathcal{O}(n)$ time. Like probing the average case is dependent on the nature of the collisions of the keys which depends on the specific keys stored in the map.

At first glance, it seems like we do not have resizing issues with chaining like we do with probing. With lists holding each bucket, we never run out of room for the keys. However, because our arrays have finite size, if we store many more keys that we have buckets, the pigeonhole principle tells us we will have collisions. In other words, eventually our map will look as follows:

```
[ ][ ][ ][ ][ ]
 |  |  |  |  |--> [ ... ]
 |  |  |  |--> [ ... ]
 |  |  |--> [ ... ]
 |  |--> [ ... ]
 |-->[ ... ]
```

where each of the buckets contain many keys. In this situation, it is prudent to *resize* the array, creating more buckets and subsequently more opportunities to spread out keys among the different buckets.

Like probing, we simply cannot create an array of double the size and copy over the elements blindly—we will not respect the new preferred indices of the keys if we do this. Instead, we must create an array with double the size and then *rehash* the key-value pairs back into the newly created array.

When we choose to resize the array with chaining is different than probing. With probing, we can simply resize the array when it is full. With chaining, we never truly "run out of space"; instead, we must choose an appropriate *load factor* that consists of the ratio of the size of the number of entries n to the number of buckets k: n/k. A load factor of 0 indicates that the map is empty. In contrast, a load factor of 1 indicates the map is "full" in the sense that adding any more elements guarantees a collision. With a low load factor, the backing array is sparsely populated meaning that we are likely wasting space with empty array indices without positively affecting our lookup times. A high load factor means that there will likely be many collisions pushing our lookup time to $\mathcal{O}(N)$.

12.1.3 Removal

One final operation that we have yet to discuss is *removal* of key-value pairs. In the case of chaining, removal amounts to removing a key-value pair from an association list which is easy to do. Like an array list, we can choose to leave the backing array sparsely populated or *contract* it to increase its load factor, saving space in the process.

With probing, removal becomes a more nuanced affair. Consider the following array:

```
[k1, k2, k3, k4]
```

Where k2 and k3 both have id 1, and suppose we now remove k2. Because an array must contain a value in each of its indices, we can elect to null out index 1.

```
[k1, null, k3, k4]
```

However, what happens when we lookup `k3` now? We'll start at its preferred index, 1, and note that no element is there. We could pass over index 1 and find k3 at index 2. But now consider this scenario where the map is empty:

```
[null, null, null, null]
```

Here, when we look up at index 1, we note that no element is there. Rather than skipping over to the next element, we want to terminate the search *right now* so we don't spend $\mathcal{O}(n)$ time trying to find an element in this empty map.

Thus, we need to differentiate between an *empty index due to a deletion* and an *empty index due to no key-value pairs being added there yet*. In the former case, we want to move onto the next index while performing lookup; in the latter case, we do not. To obtain this behavior in Java, we may use the Java 8 `Optional<T>` class. Instances of `Optional<T>` either contain a value of type T or nothing. We can use this class as follows:

- A `null` value in the array means there is no corresponding entry in the map.
- An empty `Optional<T>` value corresponds to a deleted value.
- A non-empty `Optional<T>` value corresponds to an actual entry in the map.

12.2 Transforming Values to Integers

Finally, we need to lift the restriction on our integer maps that our keys must be values. To do this, we develop the fundamental idea of *hashing*, transforming an arbitrary value into an integer. The resulting data structure that we obtain, the *hash map*, gives us the constant-time map operations that were seeking over any data type for which we can write a *hash function*.

12.2.1 The Hash Function

A hash function is simply a function $h : T \longrightarrow \texttt{int}$ that converts a value of type T into an integer with some restrictions. To see what these are, let's consider writing a hash function for type char (so $T = \texttt{char}$). Here is a simple example of a hash function:

$$h(c) = 0$$

which transforms every character c into the integer 0. This is not a very good example of a hash function because it maps every character into the same index! For example, if we used a chaining scheme and our keys were characters, using this hash function to convert characters into indices nets us the following structure:

```
[ ][ ][ ][ ][ ]
 |
 |--> [c1, c2, c3, ...]
```

Where all the keys sit in the zero bucket! We say that this hash function does not *distribute* its keys among the space of possible indices effectively.

On the other hand, the following hash function:

$$h(c) = \text{rand}(0, 100)$$

effectively distributes its keys—it indeed chooses a random index in the range 0 through 100 on each invocation. However, the resulting hash value is not *consistent* meaning that two separate calls to $h(c)$ will likely result in two different indices. This makes lookup with such a hash function impossible!

In summary, we need a hash function that:

1. Distributes its keys as evenly as possible among the space of possible integers.
2. Consistently assigns keys so that if $k1 = k2$ then $h(k1) = h(k2)$.

Note that the second requirement says that equal keys produce equal *hash values*. The converse is not necessarily true! A valid hash function may have $h(k1) = h(k2)$ but $h1 \neq h2$. This is precisely a *key collision* that we discussed previously and have methods for resolving after-the-fact. The first hash function we examined—the constant hash function—is valid, but it produces too many collisions.

For characters in Java, the hash function is quite simple: take the character value of the `char`!

```
public static int hashChar(char ch) {
    return (int) ch;
}
```

Note that the way that we represent characters in a computer program is by assigning a unique integer to every character (known as the Unicode character encoding standard). Thus, we can simply use this integer as an index into our backing array for keys that are characters.

The function above is certainly consistent as it always returns the same value for, *e.g.*, `'a'`. It is *very* well distributed because by definition of the Unicode standard, every character has a unique integer value. We call such a hash function that provably never produces any collisions between elements a *perfect hash function*.

12.2.2 Hash Functions for Primitives

We can apply similar logic to the different primitive types of Java in order to obtain hash functions for each. Recall that the primitives types in Java are:

- `int`
- `char`
- `boolean`
- `float`
- `long`
- `double`

For integers, the hash function can simply be the identity function:

$$h_{\texttt{int}}(n) = n$$

As discussed, the hash function for characters simply takes their character value as the hash:

$$h_{\texttt{char}}(c) = (int)\ c$$

For booleans, there are only two possible values—`true` and `false`—we can simply assign two integers to them.

$$h_{\texttt{bool}}(b) = \text{if } b \text{ then } 1 \text{ else } 0$$

Floats are trickier to hash. Recall that floats and doubles are floating-point numbers—decimals of finite length—represented using the IEEE floating point standard. As a first attempt, we might try to simply truncate the float to an integer via a cast. However, for certain sets of keys, this is a very bad hash function. For example, if all our keys are drawn from the rationals in the range $(0, 1.0)$, then this function will hash them all of them to the zero index in our map.

A better hash function comes from the realization that a float is the same size as an integer—32 bits. Because of this, there is a one-to-one correspondence between floats and integers (although it is not a natural correspondence because a float is represented differently than an integer). In Java we can take advantage of this correspondence by using the static method `Float.floatToIntBits` to convert a float into an integer by reinterpreting the 32 bits of the float as an integer. For the float 3.14 this yields the integer 1078523331. In contrast, if we simply truncate-cast the float to an int, we receive 3.

Like `char`, this hash function for `floats` is also perfect because it maps every possible `float` to a unique integer. In contrast, we cannot do the same thing for longs. This is because a `long` is 64 bits whereas an `int` is 32 bits. Because there are many more `long`s than `int`s, there will be collisions with any hash function that we can design.

There are a variety of possible hash functions we can use, *e.g.*,

- $h_1(f) =$ `(int) (f \ Integer.MAX_INT)`: mod the `long` by the maximum integer.
- $h_2(f) =$ `(int) (f >>> 32)`: use the most significant 32 bits of the `long`.
- $h_3(f) =$ `(int) (f << 32 >>> 32)`: use the least significant 32 bits of the `long`—shift the least significant bits to the top, then shift back.

Each of these hash functions are consistent, so they are valid hash functions. However, there are potential problems with each:

- With h_1, the modulo operator is a costly operation which we would like to avoid with our hash function if possible.
- With h_2, Shifting is cheap, however, we ignore the bottom 32 bits of the `long`. If all our keys only differed in the bottom 32 bits, then this hash function would send every key to the same hash value, zero.
- Likewise with h_3, if the all our keys differed in the top 32 bits, then this hash function would also send every key to the same hash value.

To avoid these difficulties, Joshua Bloch in his book, *Effective Java*, recommends the following expression for its hash function:

$h(f) =$ `(int) (f ^ (f >>> 32))`

This is a variation of using the most significant bits of the `long` that takes into account the least significant bits as well. It does this by performing a bitwise or with the top 32 bits (the result of the shift) and the bottom 32 bits. This is also what the Java standard library performs for its hash function for `long`s. Again, note that this is not a perfect hash function, but it performs well in practice.

Finally, to hash a `double`, we can simply combine our strategies for a `float` and `long`:

1. Convert the `double` to a `long` by reinterpreting its 64 bits (using `Double.doubleToLongBits`).
2. Perform the `long` hash function described above to produce an `int`.

12.2.3 Hash Functions for Objects

Of course, we are unlikely to need to only produce hash values for primitives. More likely, we will need to hash object values of a particular class type that we have designed that has arbitrary fields. In Java, the `Object` class provides a method:

```
/** @return a hash value appropriate for this object */
public int hashCode() { /* ... */ }
```

That any class can override to provide appropriate hashing behavior for their instances. On top of requiring that an implementor's `hashCode` function must return an integer, Java requires several other properties of such an implementation:

- Like the hash functions discussed previously, `hashCode` must be *consistent*—it must report the same hash value for any particular object during the execution of the program.
- If two objects are deemed equal (via the `equals()` method), then their hash values must be identical.

Like hash functions, `hashCode` is free to return the same value for un-equal objects—this is a hash collision and ideally avoided as much as possible.

The default implementation of `hashCode()` on most versions of Java returns the *memory address* of the object. Note that this is a unique representation of the object as only one object can sit at that address in memory. This is also the default implementation of `equals()`—using *object identity* as the definition of equality. However, we frequently want to override the definition of equality as *structural* over the fields of the object. Likewise, to meet condition (2) described above, we must implement `hashCode` to perform similarly. This is an important enough rule to codify below:

If a subclass overrides the `equals` method, then it must also override the `hashCode` as well.

As an example, consider the humble Point class:

```
public class Point {
    private int x;
    private int y;
    public Point (int x, int y) { this.x = x; this.y = y; }
    public boolean equals(Object o) {
        if (o instanceof Point) {
            Point rhs = (Point) o;
            return x == rhs.x && y == rhs.y;
        } else {
            return false;
        }
    }
}
```

Equality over points is defined to be equality over their x- and y-coordinates. How do we override `hashCode` to mimic this behavior? There are two integer fields for Point—how do we combine them into a single hash value? One solution is to simply add the two fields together:

```
public int hashCode() {
    return x + y;
}
```

However, this means that the two points $(1, 0)$ and $(0, 1)$ will hash to the same value: 1. We would like to avoid *permutation collisions* of this nature since they are common for a variety of objects.

Joshua Bloch recommends the following general strategy for implementing hashCode for classes taking into account their fields:

```java
public int hashCode() {
    int result = /* a random non-zero integer */;
    for (/* each field f in this object */) {
        int c = /* the result of hashing the field f */;
        result = 31 * result + c;
    }
    return result;
}
```

Start with a random non-zero integer. Then for each field of the object, add its hash code to the result after multiplying the current value of result by 31. This approach is essentially taking the *linear combination* of the hash codes of the fields of the object with the exception of multiplying the accumulated value by 31. According to Bloch, the value 31 is chosen because:

1. 31 is an odd prime. An odd prime is chosen to minimize the probability that the multiplication does not result in the hash value and the number of buckets having a similar prime factorization. If this is the case, then the hash values will collide much more frequently.
2. 31 is specifically chosen because the compiler can turn operations over this number into series of shifts and subtractions, specifically `31 * i == (i << 5) - i`. Although, any odd prime of sufficient size is fine.

For our point class above, we would adopt this strategy into its `hashCode` method as follows:

```java
public int hashCode() {
    int result  = 1091;
    result = 31 * result + x;
    result = 31 * result + y;
    return result;
}
```

Chapter 13

Priority Queues

Imagine that you are writing a system to manage tickets—service requests—for an IT department. A natural choice of data structure to hold these tickets is a *queue* which provides *first-in-last-out* behavior. Recall that queues provide two important operations:

- `enqueue(T v)`: adds the value `v` onto the end of the queue.
- `dequeue()`: removes and returns the oldest value (the first value in line) from the queue.

This interface is suitable for dealing with the line-like-behavior that our system needs to manage.

However, a queue is insufficient if the tickets also carry along *priorities* with them. Some tickets are higher priority than others—an electrical fire should take precedence over an accountant's email being down—and our system needs to be able handle this. In particular, we ought to require that dequeue return the *highest priority item* in the queue, regardless of how long it has been in the structure. If there are multiple items with the same highest priority, then we can service any of them first.

13.1 The Priority Queue ADT

A priority queue is an abstract data type that represents a queue-like structure that respects priorities. A priority is defined to be an *ordering* on the elements contained in the queue. For example, the tickets may have a numeric value associated with them denoting their priority, and we could order the tickets based off this priority.

A priority queue provides the following operations:

- `void add(T v)`: adds the value `v` into the queue, respecting priority order (*i.e.*, enqueue).
- `T poll()`: removes and returns the value with the highest priority from the queue (*i.e.*, dequeue).
- `T peek()`: returns, but does not remove, the value with the highest priority in the queue.

If multiple values have the same priority, then the order in which they are dequeued is unspecified. For example, consider a priority queue for tickets as described above where a ticket is represented by an integer that is its priority. For simplicity's sake, we'll represent a ticket by an integer representing its priority with higher priority tickets needing to be serviced first. For example, if we start with an empty priority queue and add in tickets 10, 5, and 2, the queue has the following shape:

$$\text{(front)} \quad [10, 5, 2].$$

Peeking at the top element of the queue will result in 10 because it is the highest priority ticket. After adding tickets 15 and 7, the queue changes to:

$$[15, 10, 7, 5, 2].$$

Polling the queue at this point dequeues 15 from the front of the queue, leaving:

$$[10, 7, 5, 2].$$

Finally, if we add another ticket with priority 5, the queue becomes:

$$[10, 7, 5_a, 5_b, 2].$$

The newest ticket of priority 5, denoted 5_b appears after the older ticket, 5_a, in the queue.

Note that the higher priority ticket goes to the front of the queue. For simplicity's sake, we show the queue as an ordered list although the data structure we use to represent this priority queue may not maintain this ordering. The only thing it needs to do is ensure that the highest priority element is easily removable when the queue is polled.

13.2 Heaps

Our example suggests a simple implementation: an ordered array list where elements are ordered by their priority.

- `void add(T v)`: because the list is ordered, we can use binary insertion to insert `v` into the list in amortized $\mathcal{O}(\log n)$ time.
- `T peek()`: because the list is ordered, the head of the list is the highest priority. We can access this element in $\mathcal{O}(1)$ time.
- `T poll()`: poll requires that we remove the head of the list which takes $\mathcal{O}(n)$.

Can we do better than this? Recall that a *binary search tree* allows us to maintain an ordered list but with $\mathcal{O}(\log n)$ time complexity for `add` and `remove`. This sounds good on paper, but the problem is that the $\mathcal{O}(\log n)$ time complexity is dependent on the BST being *balanced*. If it is balanced, then we obtain the desired $\mathcal{O}(\log n)$ complexity. However, if the tree is degenerate, *i.e.*, a linked list, then we have $\mathcal{O}(n)$ time complexity instead.

There are general schemes for maintaining a balanced tree that allow us to get the $\mathcal{O}(\log n)$ complexity we want but at the cost of higher constant factors in the runtime. However, can we obtain similar performance in this restricted domain of supporting a priority queue without the complexities of a general balancing strategy?

It turns out that we can do this by relaxing the invariant on our binary search tree. Recall that the *binary search tree invariant* says that for any node in a binary search tree:

(a) The left branch contains elements that are less than the element at this node.

(b) The right branch contains elements that are greater than the element at this node.

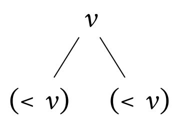
Because of this invariant, we are forced to unbalance the tree in certain situations, *e.g.*, inserting elements in ascending order. If we relax the invariant, we can hit a sweet spot between enforcing the ordering that we to support a priority queue while allowing us to easily balance the tree.

13.3 The Heap Invariants

The data structure we'll use to efficiently implement a priority queue is a called a *heap* (which has no relation to the heap in memory). A (binary) heap is a tree which maintains two invariants:

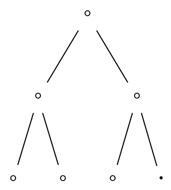
- The *semantic binary heap invariant* says that for any node in the tree, the sub-trees of that node only contain elements *less than* the element at this node.
- The *structural binary heap invariant* says that the heap is always *complete*. That is, all the levels but the last of the heap are completely filled in, and the last level is filled from left-to-right.

The semantic invariant is represented graphically as follows:



This may seem like a useless property, but by doing this, we implicitly require that all elements greater than v appear above it in the tree. By applying this reasoning recursively at each level of the tree, we know that the *maximum priority element* must be placed at the root of the tree.

The structural invariant is represented graphically as follows:



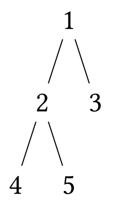
Note that with such a tree, the length of any path from the root to a leaf is upper-bounded by $\log n$ which is critical in ensuring that the runtime of our operations will be $\mathcal{O}(\log n)$.

A final note on the semantic invariant of our heaps: by choosing to push smaller elements further down the tree, the maximum element sits at the root. We call such a heap a *max heap*. In contrast, we could instead have our semantic invariant require that the children of a node only contain elements *greater than* the value at the node. By doing this, the minimum element sits at the root of tree. Such a heap is a *min heap*. For the purposes of simplifying our discussion, we will consider a max heap in our discussions below. However, keep in mind that a min heap is obtainable by simply flipping the ordering in our invariant.

13.4 Array-based Trees

Because our heaps are complete trees, we are able to use an *array* to represent the tree rather than a linked structure (compare array lists versus linked lists). The array will contain the contents of the nodes of our tree, and rather than containing references to its children, we will use explicit formulae to find the indices of children and parent nodes given the index of a particular node in the tree.

To arrive at these formulae, we note that a natural way to lay out the elements of a complete tree in an array is to proceed in a top-down, left-to-right fashion. For example, if we have the following tree:



We could represent it with the following array:

$$[1, 2, 3, 4, 5, \ldots]$$

keeping in mind that like an array list, only part of the overall array is in use at any given time. Because the tree is complete, this layout strategy ensures we fill the array from left to right. This fact is why we did not previously consider using an array to represent a tree. Most trees will not be complete like a heap and so, there will be many indices of the array that are unused.

If we look at each node in the array:

- 1 is at index 0 with children 2 (index 1) and 3 (index 2)
- 2 is at index 1 with children 4 (index 3) and 5 (index 4)
- 3 is at index 2 with no children (indices 5 and 6).

From this, we can derive the following formulae:

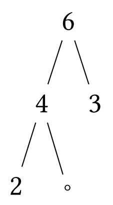
- The *left child* of the node at index i is $i \times 2 + 1$.
- The *right child* of the node at index i is $i \times 2 + 2 = (i + 1) \times 2$.
- The *parent* of the node at index i is $\lfloor \frac{i-1}{2} \rfloor$ for nodes that are not the root of the overall tree.

13.5 Heap Operations

Now we discuss implementing each of our heap operations in terms of our array-based tree data structure:

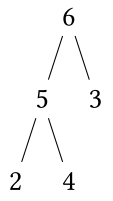
Peek Noting that the root of the tree is the first index of the array, we simply return that element. This takes $\mathcal{O}(1)$ time to do so.

Add Adding an element requires a bit more thought than `peek`. We must add an element in such a way that the maximum element is the root of the tree. When we add a new smallest element into the heap, we can simply insert the element in such a way as to maintain a complete tree, *i.e.*, top-down and left-to-right. For example, if we insert 1 into the following heap:



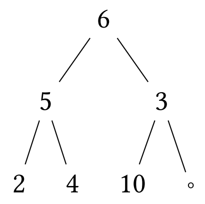
Then we can make 1 the right-child of 4 which preserves both the semantic and structural heap invariants.

However, what happens if we need to add an element that won't be a leaf in the tree? For example, if we insert 5 into the tree, it must be in either the left or right subtree of 6, but it must also be the parent of 4 and 3. Because our semantic invariant is relaxed, we have a choice about which branch of the tree we should modify. To preserve the structural invariant, it makes sense to make 5 the direct left child of 6 and make 4 a right-child of it.



What happens if we add a element that displaces the root of the tree, *e.g.*, 10? Where should 6 go, and how should its children be shifted to maintain both invariants?

Rather than worrying about preserving both invariants at once, we'll place a new element in the tree in such a way as to preserve one of the invariants immediately and then fix its position in the heap to fix the second. It is relatively easy to place a new element in the heap to obey the structural invariant: just respect the top-down, left-to-right order we've discussed.

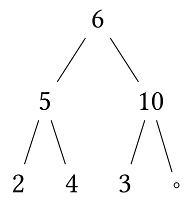


Note that this is particularly elegant with our array-based implementation of a heap. Insertion that preserves completeness is simply adding onto the end of the array:

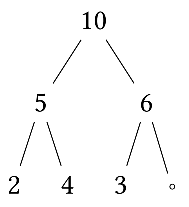
$$[6, 5, 3, 2, 4, 10, \ldots]$$

One downside is that like an array list, we will need to grow our backing array, *e.g.*, double its size, when it is full. We also need to make an amortized analysis arguing that this growth operation is irrelevant when considering sequences of add operations. All of these details are identical to those discussed early with array lists, so we won't discuss them here in favor of focusing on details specific to heaps.

With our element inserted into the heap, we must fix up its position to maintain the semantic invariant. To do this, note that in our heap diagram above, 10 is out of position with respect to its parent, 3. We can fix this by simply swapping 10 with 3.



Now 10 is out of position with respect to its new parent, 6. To fix this, we swap them, leaving 10 in its final position at the root of the tree.



This fix-up operation is called *sifting* or *percolation*, specifically *sift up* and *percolate up*. We repeatedly push the inserted element up the heap until it is in a position where the semantic invariant is preserved. Note that because the element that we swap with its parent is always greater than its parent, the semantic invariant over the other branch is preserved, ensuring correctness of the operation.

Thus, addition into the priority queue with a heap has the following steps:

(a) Add the element onto the end of the heap.

(b) Sift/percolate up the element into its proper position in the array.

The first operation takes (amortized) $\mathcal{O}(1)$ time as it is equivalent to array list addition (to end of the list). In the worst case, the sifting operation takes $\mathcal{O}(\log n)$ time because the inserted element may be sifted from leaf to root in the tree which requires a number of swaps equal to the height of the tree. Thus, overall the runtime of add is (amortized) $\mathcal{O}(\log n)$.

Poll Analogously to `add`, we have the issue that by removing the root of the tree, we end up potentially disrupting both invariants. To get around this, we preserve the structural invariant by moving the last element of the priority queue to the root (preserving structure) and then *sift down*/*percolate down* the element into its proper position in the priority queue.

Sift down operates analogously to the sift up operation described for add. The only wrinkle is that we have a choice of whether we sift down the left-hand branch or the right-hand branch of the node. Either leads to a correct implementation, however, to minimize the number of potential swaps we need to perform, we should favor the branch with the larger element, increasing the likelihood that we find an element that is less than this one.

Summarizing the poll operation:

(a) Replace the root with the last element in the heap.

(b) Sift/percolate down the new root into its proper position in the array.

(c) Return the old root value.

Like add, the first step takes constant time and the sifting takes $\mathcal{O}(\log n)$ time for an overall time complexity of $\mathcal{O}(\log n)$.

13.6 Heap Sort

With our heap structure defined, we can readily implement a sorting algorithm using it:

(a) Insert the elements into a min-heap.

(b) Repeatedly poll the heap until it is empty, reading off the element in ascending order in the process.

Insertion takes $\mathcal{O}(n \log n)$ time because for each of the n elements, we perform an $\mathcal{O}(\log n)$ add. Likewise, we require n calls to poll which takes $\mathcal{O}(n \log n)$ time overall. Thus, the overall run time of *heap sort* is $\mathcal{O}(n \log n)$, same as merge sort and quicksort. The constant factors associated with heap sort are higher than quicksort, in particular, the need for an auxiliary array to hold the heap. However, unlike quicksort which has worst case $\mathcal{O}(n^2)$ time for bad pivots, heap sort has consistent $\mathcal{O}(n \log n)$ performance which may be desirable depending on the problem at hand.

Part IV

Large-Scale Program Design

Chapter 14

Inheritance

So far, we have used interfaces to define the *contract* between an abstract data type and its implementation. In the context of Java's type system, interfaces also induce a *subtyping* relationship between interface and implementor type where we say that the implementor type *is a* interface type. This allows us to substitute an instance of the implementor where ever the interface type is required. We called this flexibility *(subtype) polymorphism*: the ability to write code that works with many types.

Interfaces are a powerful, fundamental mechanism for establishing such a subtyping relationship. However, they carry with them an important limitation: we cannot provide any implementation to the implementor of the interface. While the intent of an interface is that implementors provide different implementation, sometimes we either want to (1) share parts of the implementation or (2) provide some default implementation that implementors are free to *override* if desired. To do this, we require another mechanism for *sharing implementation* in addition to establishing an *"is a"* relationship between types.

14.1 An Example: Shapes

Suppose that we need to represent a collection of shapes such as squares, circles, and triangles in our program. We might design an interface that represents a shape:

```
// Shape.java
/**
 * A shape, e.g., a square, circle, or triangle.
 */
public interface Shape {
    /**
     * @return the x-coordinate of this shape
     */
    public int getX();

    /**
     * @return the y-coordinate of this shape
     */
    public int getY();

    /**
```

```
     * @return the width of this shape
     */
    public int getWidth();

    /**
     * @return the height of this shape
     */
    public int getHeight();

    /**
     * @return the area of this shape
     */
    public int getArea();
}
```

As well as provide some implementations of this interface:

```
// Rectangle.java
public class Rectangle implements Shape {
    private int x;
    private int y;
    private int width;
    private int height;

    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

    public int getX() { return x; }
    public int getY() { return y; }
    public int getWidth() { return width; }
    public int getHeight() { return height; }
    public int getArea() { return width * height; }
}

// Circle.java
public class Circle implements Shape {
    private int x;
    private int y;
    private int radius;

    public Circle(int x, int y, int radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }

    public int getX() { return x; }
    public int getY() { return y; }
    public int getWidth() { return 2 * radius; }
    public int getHeight() { return 2 * radius; }
    public int getArea() { return (int) (Math.pi * (Math.pow(radius, 2))); }
}
```

Naturally, Square and Circle have different sorts of representation as they are different shapes. However, they have something in common: the representation of their position as a coordinate pair. Furthermore, the implementation of this coordinate pair is the same—two fields and two getters for each of the components of the pair. Because an interface does not let us provide implementation details, we have to duplicate this code in Square and Circle which is undesirable.

Instead of creating an interface to represent the abstract data type of a shape, we'll instead create an *abstract class* for this purpose:

```
/**
 * A shape, e.g., a square, circle, or triangle.
 */
public abstract class Shape {
    private int x;
    private int y;

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

    /**
     * @return the x-coordinate of this shape
     */
    public int getX() { return x; }

    /**
     * @return the y-coordinate of this shape
     */
    public int getY() { return y; }

    /**
     * @return the width of this shape
     */
    public abstract int getWidth();

    /**
     * @return the height of this shape
     */
    public abstract int getHeight();

    /**
     * @return the area of this shape
     */
    public abstract int getArea();
}
```

A class marked `abstract` sits between an `interface` and a `class`:

- An abstract class cannot be instantiated. In other words, the expression `new Shape(0, 0)` will give a compiler error complaining that you are trying to instantiate an abstract class. This makes sense because `Shape` is an "abstract" entity, *i.e.*, shapes do not exist but instead there are squares, triangles, circles, *etc.*.
- Because an abstract class is a class, it can provide implementation like a normal class.
- An abstract class can optionally mark methods as `abstract`. An `abstract` method does not have an implementation; a class that *extends* this abstract class will need to provide an implementation of this method.

Like interfaces, we can declare a subtype relation between an abstract class and a normal class. However, rather than *implementing* an interface, we say that the (normal) class *extends* the abstract class.

```
// Rectangle.java
public class Rectangle extends Shape {
    private int width;
    private int height;

    public Rectangle(int x, int y, int width, int height) {
        super(x, y);
        this.width = width;
        this.height = height;
    }

    public int getWidth() { return width; }
    public int getHeight() { return height; }
    public int getArea() { return width * height; }
}

// Circle.java
public class Circle implements Shape {
    private int radius;

    public Circle(int x, int y, int radius) {
        super(x, y);
        this.radius = radius;
    }
    public int getWidth() { return 2 * radius; }
    public int getHeight() { return 2 * radius; }
    public int getArea() { return (int) (Math.PI * (Math.pow(radius, 2))); }
}
```

By extending the Shape abstract class (using the `extends` clause on the class declaration), both Circle and Rectangle *inherit* the state and behavior defined by `Shape`. We say that Shape is the *parent class* or *super class* of `Circle` and `Rectangle`. Conversely, we say that Circle and Rectangle are *subclasses* of Shape. Namely, they both possess two fields, `x` and `y` of type `int` and two getter methods `getX()` and `getY()`. To extend the `Shape` class, `Rectangle` and `Circle` must provide implementations for the abstract methods `getWidth()`, `getHeight()`, and `getArea()`.

However, since we inherit these fields, how do we initialize them? Note that because `x` and `y` are both declared as `private` in `Shape`, we cannot access them from `Square` or `Circle`. This does not mean we do not inherit them, it just means that their *names* are not visible from within `Square` or `Circle`. To initialize these fields, we must access the constructor of the `Shape` class. To do this, we use a *super class constructor invocation* as the first line of the constructors of `Circle` and `Rectangle`. The syntax of a super class constructor invocation is:

```
super(<expr>, ..., <expr>);
```

You can think of `super` as standing in for the name of the super class. For example, in `Circle`, we invoke the `Shape` constructor that takes values to initialize `x` and `y` with the call `super(x, y)`.

By extending `Shape`, `Rectangle` and `Circle` are both considered subtypes of `Shape`, just like with an interface. In particular, the following code snippet works as you'd expect:

```
Shape shape1 = new Square(0, 0, 100, 100);
Shape shape2 = new Circle(10, 10, 100);
System.out.println(shape1.getArea());    // 100 * 100
System.out.println(shape2.getArea());    // pi * 100^2
```

We can also extend normal classes, not just abstract classes. For example, if we wish to specialize `Rectangle` further to a `Square`, we can extend `Rectangle`:

```
public class Square extends Rectangle {
    public Square(int x, int y, int length) {
        super(x, y, length, length);
    }
}
```

This might seem redundant; we could simply create a square by specifying a `Rectangle` with equal width and height. However, by encoding a `Square` as a class, the type checker can now ensure that the user provides squares when necessary. Thus we turn what would otherwise be runtime errors into compile time errors.

14.2 Restrictions on Class Inheritance

After walking through this example, it seems like class inheritance is strictly more powerful than interface extension. In addition to providing `abstract` methods, we can also provide implementation—both fields and methods—with an abstract class. So what's the downside?

The downside is that a class may only *extend a single class*, but it may *implement multiple interfaces*. Java elects to be a *single-inheritance object-oriented programming language* because it turns out there are lots of complications involved with *multiple inheritance*, *e.g.*, what happens if two super classes provide a field or method with the same name? Rather than dealing with these problems (the *diamond problem*), Java tries to keep things simple by requiring that a class can only inherit from at most one super class.

14.3 Visibility and Dynamic Dispatch

Previously, we explored a particular case study of using *inheritance* over interface implementation: we want to share some implementation details between subclasses. It turns out that this one addition complicates the mechanics of our program greatly. We'll now turn our attention to looking at this mechanics in detail, in particular, three language features:

1. Visibility with subclasses.
2. Accessing and invoking subclass state and behavior.
3. Overriding subclass behavior.

14.4 Visibility with Subclasses

Recall that a member marked `private` was only accessible to code *within* the declaring class, for example,

```java
public class C {
    private int x;
    private int foo() { return 0; }
    public C() { this.x = 5; }
    public static void test() {
        C c = new C();
        System.out.println(c.x);      // 5
        System.out.println(c.foo());  // 0
    }
}

class Bad {
    public static void test() {
        C c = new C();
        System.out.println(c.x);      // error: x not visible
        System.out.println(c.foo());  // error: foo not visible
    }
}
```

We use privacy modifiers to hide implementation details from clients of a class. However, subclasses (established with inheritance and `extends`) sit somewhere between "client" and "author". On one hand, they are a client of their superclass in the sense that they are using its functionality. On the other hand, it is plausible that they will need direct access to some of these implementation details, *e.g.*, to update a field based on the subclass's refine behavior.

Therefore, we need a refined accessibility modifier that allows subclasses to access superclass members without exposing those members to everyone. This accessibility modifier is `protected`. For example, consider modifying a subset of our Shape class from before with `protected` members:

```
public abstract class Shape {
    protected int x;
    protected int y;

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

    /**
     * @return the x-coordinate of this shape
     */
    public int getX() { return x; }

    /**
     * @return the y-coordinate of this shape
     */
    public int getY() { return y; }
}
```

Now, our `Rectangle` subclass can modify the `x` and `y` fields directly:

```
public class Rectangle {
    // ...
    public void translate(int dx, int dy) {
        this.x += dx;
        this.y += dy;
    }
}
```

So why not make all our variables `private` instead of `protected`? Again, this becomes a question of *least priviledge*. Exposing, in particular, a field allows us to perform two operations:

1. Writing to the field.
2. Reading from that field.

If a superclass does not wish to allow subclasses both of these behaviors, then the field should be `private` instead of `protected`. This is desirable because if there's some *invariant* that exists between the superclass's fields (*e.g.*, in our `ArrayStack` implementation, that the `top` field always pointed to the top of the stack in the `data` array), we would like to not give a subclass the opportunity to mess it up.

As a rule of thumb, if you predict that your subclasses will not need to both read from and write to a field (and you do not need to "wrap" the reading/writing, *e.g.*, to perform a pre-condition check), then that field should be marked `private` rather than `protected` to prevent this behavior. This is the same logic that you should apply to determining if a field ought to be `public`. In practice, this means few of your fields will be marked `protected` or `public` because you will rarely want to expose fields in this manner.

14.5 Overriding Subclass Behavior

One of the important benefit of extending a class is to inherit its (public and protected) fields and methods. However, what if we wish to *override* the behavior of one of these methods? The canonical example is a superclass that provides some default behavior with the intention that a subclass provide more refined behavior. As a toy example, consider a small class hierarchy for animals.

```java
public abstract class Animal {
    private boolean alive;
    public Animal() {
        this.alive = true;
    }
    public boolean isAlive() { return alive; }
    public void makeNoise() { System.out.println("Ding"); }
}

public class Dog extends Animal {
    public Dog() { /* Implicitly calls super() */ }
}

public class Cat extends Animal {
    public Cat() { /* Implicitly calls super() */ }

    @Override
    public void makeNoise() { System.out.println("Meow"); }
}

public class Lion extends Cat {
    public Lion() { /* Implicitly calls super() */ }

    @Override
    public void makeNoise() { System.out.println("Roar"); }
}
```

Here, we have a *class hierarchy* consisting of four classes. The relationship can be drawn as follows:

```
      Animal
         |
     ---------
     |       |
    Cat     Dog
     |
   Lion
```

Where `Animal` is the top-most superclass, a `Cat` is an `Animal`, a `Dog` is an `Animal` (but is not a cat), and a `Lion` is a `Cat`. `Animal` provides a private field and public method for determining if an `Animal` is alive. `Cat`, `Dog`, and `Lion` all inherit these members.

However, Animal also specifies a method `makeNoise` with a generic sound suitable as a default for any `Animal`. The `Dog` class inherits this generic sound, so the following code produces the expected result:

```
Dog d = new Dog();
d.makeNoise();          // Ding
```

This is, of course, not the sound that a `Dog` makes. We need some way of providing our own class-specific behavior `makeNoise`. `Cat` and `Lion` both do this by *overriding* the `makeNoise` method to provide their own implementation. For example, the following code:

```
Cat c  = new Cat();
Lion l = new Lion();
c.makeNoise();          // Meow
l.makeNoise();          // Roar
```

produces refined behavior for `makeNoise` based on the type of the object we call the method on: “Meow” for cats and “Roar” for lions. To override a method in a subclass, we simply include an implementation of that method in the subclass. However, we also include an *annotation* on this method, `@Override`, to note that this method declaration intentionally overrides the `makeNoise` method declared in the superclass. (Note that the `@Override` annotation is a new addition in Java 1.5 and is technically optional. However, you should always use `@Override` to be explicit when you are overriding superclass behavior.)

Method overriding seems straightforward. However, things quickly become less clear when we combine method overriding with subtyping. Consider the following variable declarations, method calls, and their results:

```
Animal a1 = new Dog();
Animal a2 = new Cat();
Animal a3 = new Lion();
Cat c = new Lion();
a1.makeNoise();          // Ding
a2.makeNoise();          // Meow
a3.makeNoise();          // Roar
c.makeNoise();           // Roar
```

Note that even though the type of the variable is `Animal` in the first three cases, we use the *actual type of the object* assigned to the variable to determine which method to call. We distinguish between the two sorts of the types accordingly:

- The *static type* of a value is the type of the value as it is known to the compiler. In the case of variable declarations, this is the type of the declared variable. The static type determines what is the allowable set of methods we can call on an object.
- The *dynamic type* of a value is the actual type of the object. The dynamic type determines what implementation of a method we actually invoke.

For example, for the above code, here are the static and dynamic types for the various combinations of variable declarations and assignment that our subtyping relationships allow:

```
Animal a1 = new Dog();    // static type: Animal, dynamic type: Dog
Animal a2 = new Cat();    // static type: Animal, dynamic type: Cat
Animal a3 = new Lion();   // static type: Animal, dynamic type: Lion
Cat c1  = new Cat();      // static type: Cat, dynamic type: Cat
Cat c2 = new Lion();      // static type: Cat, dynamic type: Lion
Lion l = new Lion();      // static type: Lion, dynamic type: Lion
Dog d = new Dog();        // static type: Dog, dynamic type: Dog
```

We can use this information to determine which of the method implementations actually fire for any call to `makeNoise`.

14.6 The Mechanics of Dynamic Dispatch

Recall the rules for resolving overridden method calls:

1. We use the *static type* of an expression to determine whether the method call typechecks.
2. We use the *dynamic type* of an expression to determine which method we actually invoke.

Note that the static type is usually a more *general type* (the supertype) than what we have and the dynamic type is the more specific type (the subtype).

This rule helps us resolve method calls. However, this resolution process becomes tricky with trickier set ups of inheritance hierarchies. To better understand the process of resolving overridden method calls, we'll study how this is implemented in Java with our mental model of computation.

14.6.1 Objects on the Heap

Recall that all objects are allocated on the heap. An object is a collection of fields along with a *tag* that says what type the object is, its dynamic type. More concretely, consider the following basic class hierarchy:

```
public class C1 {
    public int x;
    public void foo() { System.out.println("C1.foo()"); }
}

public class C2 extends C1 {
    public int y;
    @Override
    public void foo() { System.out.println("C2.foo()"); }
    public void bar() { System.out.println("C2.bar()"); }
}

public class C3 extends C1 {
    public int z;
    @Override
    public void foo() { System.out.println("C3.foo()"); }
    public void baz() { System.out.println("C3.baz()"); }
}

public class C4 extends C3 {
    @Override
    public void foo() { System.out.println("C4.foo()"); }
}
```

Instances of each of these classes look like this on the heap:

```
 -------   -------   -------   -------
 | C1  |   | C2  |   | C3  |   | C4  |
 |x| 0 |   |x| 0 |   |x| 0 |   |x| 0 |
 -------   |y| 0 |   |z| 0 |   |z| 0 |
           -------   -------   -------
```

The process of resolving overridden methods is called *dynamic dispatch*. How does the dynamic dispatch process work? The key insight is that the type tag associated with each object is in actually a *pointer* to a table containing the methods for that class. For example, for C3 we have:

```
 -------
 | C4  |------> [vtable for C4]
 |x| 0 |
 |z| 0 |
 -------
```

This table of methods is called a *virtual table*, or vtable, owing from its C++ roots where overridable methods are annotated with the virtual keyword. The vtable for a class contains an entry for every overridden method in the class as well as a pointer to the superclass's vtable:

```
 --------------          --------------          --------------
 | C4 vtable  |     ---> | C3 vtable  |     ---> | C1 vtable  |
 --------------     |    --------------     |    --------------
 |super vtable| ----|    |super vtable| ----|    |super vtable|
 |  C4.foo()  |          |  C3.foo()  |          |  C1.foo()  |
 --------------          |  C3.baz()  |          --------------
                         --------------
```

Suppose that we have a variable declaration and method invocation:

```
C1 c = new C4();
c.foo();
```

How does dynamic dispatch proceed? We first note that this code *typechecks and compiles* because C1, its static type, declares a foo() method. Then, at runtime, we:

1. Follow the vtable pointer of the object we invoke the method.
2. Lookup the desired method in the vtable.
3. If the method exists, we invoke that method. Otherwise, we follow the superclass vtable pointer and repeat the process.

For the above code, this amounts to looking at the vtable pointer for the C4 object. In the corresponding table, we find an entry for C4.foo() so we invoke this version of foo(). In contrast, consider this method invocation instead:

```
C1 c = new C4();
c.baz();
```

We follow the vtable pointer for the C4 object and look up an entry for `baz()`. We do not find such an entry, so we follow the superclass vtable pointer to `C3`'s vtable. We then look for a `baz()` entry, find such an entry, `C3.baz()`, and invoke that method.

As an exercise it's a good to consider all of the possible combinations of variable assignments we can have with our subtyping relationship and invocation of methods on those variables. Those combinations should either result in a *type error* at compile time or the invocation of one of the versions of these methods. Try writing out these various combinations and use this mental model of dynamic dispatch to predict the result.

14.6.2 The Object Class

While `C2`, `C3`, and `C4` above have superclasses, it appears that `C1` does not. However, this turns out to be untrue; `C1` has an implicit superclass called `Object`. `Object` is the *root* of the class hierarchy in Java—all objects are ultimately subclasses of the Object class. Because of this, we say that Java has a *unified class hierarchy* where all classes have a common base class.

The `Object` class provides a number of methods available to all objects in Java. These include the familiar `toString()` method that returns a String representation for an object. All objects in Java have this method, and it is used to implicitly convert an object to a `String`, *e.g.*, when printing it with `System.out.println`. Another important method is the `equals()` method. Here is the signature for equals along with the implementation provided by Java:

```
/** @return true if this object is "equal" to the other object */
public boolean equals(Object obj) {
    return this == obj;
}
```

By default, `equals` uses *reference equality* which checks to see if the argument is the exact same object as the one we invoke equals on. If we want to determine equality between objects that we define, we ought to override equals behavior to be appropriate for the objects in question, typically *structural equality* where we check to see if the fields of two objects are equal recursively. To see this in action, consider adding an equals method for our good old Point class

```
public class Point {
    private int x;
    private int y;
    // ...
}
```

Intuitively, how should we implement equality between two points? The interpretation of structural equality for points says two points are equal if and only if their components are equal.

This is easy enough to state, but we immediately run into problems implementing this behavior in our `equals` method:

```
// in class Point
@Override
public boolean equals(Object obj) { /* ... */ }
```

The problem is that the type of the argument is `Object` so that if we tried to access the `x` field of the argument, we get a type error:

```
public boolean equals(Object obj) {
    boolean isXEq = this.x = obj.x;    // type error: Object doesn't have an x field
}
```

This is because we may call `equals` with an argument that is not a `Point`—any class is a subclass of `Object` after all.

So as a first step, we must check to see if the argument is actually a `Point` object. To do this, we use the `instanceof` operator:

```
public boolean equals(Object obj) {
    if (obj instanceof Point) {
        // compare fields
    } else {
        return false;
    }
}
```

Think of `instanceof` as a binary operator that takes an object on the left-hand side and a class name on the right-hand side. `instanceof` evaluates to `true` if the object on the left-hand side is an object that is a *subtype* of the right-hand side.

When the input is not a `Point`, we can immediately return `false`. When the input is a `Point`, we know that `obj` is really a `Point` even though its static type is `Object`. We need a way to tell the compiler "hey, I know this thing is really a `Point`—trust me". The mechanism we use to do this is a *class cast*.

```
public boolean equals(Object obj) {
    if (obj instanceof Point) {
        Point p = (Point) obj;
        return this.x == p.x && this.y == p.y;
    } else {
        return false;
    }
}
```

A class cast looks like a normal cast except that it casts objects between types. This particular kind of cast is called a *downcast* because we are casting from a more general type—`Object`—to a more specific type—`Point`. In other words, we are casting *down* the class hierarchy. The class cast makes a runtime check to ensure that expression evaluates to an object value that really is the one promised by the cast, throwing a jiClassCastException if this is not the case. However, we know by virtue of the `instanceof` check that we indeed have a `Point` object so this cast is safe.

In general, our overridden `equals` methods have the following form:

```
public boolean equals(Object obj) {
    // Check to see if input has the appropriate object type
    // Downcast to the object type
    // Perform a structural equality check on this object and the input
}
```

14.7 Is-a Versus Has-a Relationships

When designing a program in an object-oriented programming language, we decompose the program into a collection of objects. In our programs, objects are related in two primary ways:

1. *Has-a* relationships, *e.g.*, an employee has an age.
2. *Is-a* relationships, *e.g.*, an engineer is an employee.

We realize the first kind of relationship with *fields*. For example, an `Employee` class would have an `age` field. We realize the second kind of relationship with *interface implementation* or *class inheritance*. For example, we may have either an `Employee` interface or class and an `Engineer` class that implements or extends `Employee`, respectively.

How do we choose between the two sorts of relationships? In some cases, the choice is obvious just by considering the words "has-a" and "is-a". In the above examples, it seems very wrong to consider an employee an "age" or say that an engineer has an employee. In this sense, *the natural meaning of the objects* defines the relationship between objects.

However, we can concoct scenarios where we could interpret the potential relationship between objects in multiple ways. For example, we might say:

- An engineer *is an* employee.
- An employee *has a* title that is "engineer".

Both are perfectly reasonable interpretations of the relationship between "engineer" and "employee". We can either represent an engineer as a class that inherits from/implements the employee class/interface, or a "title" field of the employee class. But which of them do we choose?

When our *semantic* interpretation of our objects' relationships is ambiguous, we must resort to reasoning about the *operational* implications of our design choice. In other words, if we recognize the relationship as a field (*has-a*) or a subtype (*is-a*), what is the effect on the rest of our program? What are we able to do with one programming construct that we cannot do with the other? What difficulties do we encounter using one construct over the other?

Recall that the primary feature that both interface implementation and class inheritance provide is subtyping. That is, we can substitute a subtype (a class that implements an interface or inherits from a superclass) anywhere that a supertype is expected. This gives us the ability to write (subtype) polymorphic code—code that operates over many types. Note that this feature has to do with the (Java) type of an object. So if a disputed property could function as part of the type of an object, then this sort of question between *is-a* and *has-a* arises.

For example, we may decide that all employees receive a salary. Therefore, we can enforce this by specifying a `getMonthlySalary()` method in the Employee class. When we need to gather up the total amount of money we need to pay out to the company, we can simply call the method uniformly on each Employee:

```java
List<Employee> employees = /* ... */ ;
int total = 0;
for (Employee e : employees) {
    total += e.getMonthlySalary();
}
System.out.println("The total is: " + total);
```

Implicitly throughout this code, the subtyping relationship between `Employee` and its subclasses is enforced at compile time via typechecking. In contrast, if we had a field of type `String` denoting the title of the employee, we would need to write the `getMonthlySalary()` method of the Employee class as follows:

```java
public int getMonthlySalary() {
    if (this.title.equals("Engineer")) {
        return 10000;
    } else if (this.title.equals("CEO")) {
        return 50000;
    } else if (/* other cases */) {
      // ...
    } else  {
        return /* Some default amount... */
    }
}
```

Note that this method is very brittle. Because the case analysis is over Strings, we cannot ensure at compile time that:

1. We provide the names of the titles correctly, *i.e.*, there are no typos, and
2. We have provided cases for *all* possible titles.

In addition to substitution via subtyping, we also gain *static typechecking* by using inheritance to model this relationship. Thus if we need to take advantage of subtype polymorphism, we ought to do so via interfaces or inheritance.

However, otherwise if there's no compelling reason to use subtype polymorphism, we ought to stick with *has-a* relationships and fields, also known as *composition*. Composition has the benefit of being:

1. Simpler. We spent little time talking about the mechanics of fields because they are straightforward to understand relative to inheritance.
2. Flexible. Exploiting inheritance requires that we create classes which must be built at compile time unless we use a complicated mechanism like reflection to load classes dynamically. A field on the other hand can take easily take any value that it domain allows.

Imagine in our employee example that we did not know the full set of job titles ahead of time, *e.g.*, because we're working on a general-purpose employee management system not tied to any one corporate structure. Here, we are unlikely to be able to take advantage of typing checking because the set of possible jobs is not fixed at compile time. In this situation, it may make sense to use composition rather inheritance to represent these positions.

There is a modern maxim that you will likely hear:

Favor composition over inheritance.

This is sound advice at a surface level. Composition is less complicated than inheritance, so you should favor the simpler thing over the more complicated thing. However, as we have seen, there are specific circumstances when you want inheritance over composition: when the relationship involved can be construed as "type-defining"and the benefits of inheritance—subtype polymorphism and static type checking—make sense in context.

14.8 The Expression Problem

When we have identified a relationship between objects as being “type-like”, this raises questions of whether we identify the relationship as *is-a* or *has-a*. One particular situation where this arises is when we need to specify behavior that performs *case analysis* on a set of types. In this situation, using an *is-a* relationship via interface implementation or inheritance shines because we can really put the type system to work for us. Although sometimes, exploiting this relationship is less intuitive than you might imagine.

For example, consider the employee example that we’ve used throughout our discussion of inheritance. Suppose that we did not use inheritance and, instead, defined the type of employee by their job title, a field. And now suppose that we wanted to define a method to retrieve the salary of the employee based on their job title. The method would look like this:

```
public int getSalary(Employee e) {
    if (employee.getTitle().equals("engineer")) {
        return 100000;
    } else if (employee.getTitle().equals("chef")) {
        return 75000;
    } else if (employee.getTitle().equals("ceo")) {
        return 1000000;
    } else {
        // A default salary
        return 0;
    }
}
```

This method is unsatisfying for a pair of reasons:

1. By encoding the title as a string, we may have some flexibility in specifying jobs at runtime that we didn’t anticipate at compile time. However because of this, we lose the ability to check that we have specified the correct job title at compile time. For example, we may do something silly like misspell “engineer”.
2. Related, in addition to being unable to check for simple typos like this at compile time, we are also unable to check for *exhaustiveness*—did we cover all the possible cases of employees?

There is also a third reason why this approach is unsatisfying. In addition to typos and exhaustiveness, any errors that we introduce with this approach occur at runtime. Furthermore, the errors are *silent*. For example if we misspell or forget “engineer”, then the method return 0 which will not manifest itself as an error until we do some calculation involving the engineer’s salary. This is nearly a worst-case scenario for us! We want errors checked at compile time at the point where they occur, not checked at runtime potentially far away from their origin.

We could add inheritance into the mix. However, inheritance alone is not enough, for example, if we introduced `Engineer`, `Chef`, and `CEO` classes that extend `Employee`, we might try the following:

```
public int getSalary(Employee e) {
    if (employee instanceof Engineer) {
        return 100000;
    } else if (employee instanceof Chef) {
        return 75000;
    } else if (employee instanceof Ceo) {
        return 1000000;
    } else {
        // A default salary
        return 0;
    }
}
```

However, this code is only marginally better than the previous approach. The fact that the different jobs are encoded as types means that the compile will catch typos. However, exhaustiveness is not checked and errors are still silent.

To cover all of these bases, we must combine inheritance with dynamic dispatch. Thus, we arrive at the initial example we used to motivate dynamic dispatch:

```
public abstract class Employee {
    public abstract int getSalary();
}

public class Engineer extends Employee {
    @Override
    public int getSalary() { return 100000; }
}

public class Chef extends Employee {
    @Override
    public int getSalary() { return 75000; }
}

public class Ceo extends Employee {
    @Override
    public int getSalary() { return 1000000; }
}
```

This final approach allows us to check for typos and exhaustiveness at compile time. Note that exhaustiveness is checked because the abstract class requires that any subclass implement the `getSalary()` method. If we need to implement other operations that also perform case analysis on the possible job types, we:

1. Introduce the method in the superclass. If default behavior is required, we give a default implementation. Otherwise, we mark it abstract so that implementors are forced to override the method.
2. Override the method in each subclass with the case-specific behavior.

This sort of design—defining data by cases and then operations over that data via case analysis—is common place in computer programs. To summarize, the key features of good design in this space is:

- Extensibility of new cases.
- Extensibility of new operations.
- Statically (type) checked.

- *Separate compilation*, the ability to write new code without having to recompile existing code.

Note that the approach that we have settled on does not cover all of these desiderata. While we have static typechecking, the ability to add new cases via additional subclasses, and separate compilation when adding new subclasses, adding new operations is a bit of a bother. Every new operation we add becomes a new method in the superclass and a series of overridden methods in the subclasses. The addition of this new method requires that we recompile all of the subclasses since they have been modified to accommodate this new method. Furthermore, the logic for this operation is spread out among all the subclasses which is problematic for code understanding.

From a programming language design perspective, this problem is known as the *expression problem*. It is difficult to satisfy all four of these desiderata simultaneously and different languages (in particular, *language paradigms* such as object-oriented versus functional programming) make trade-offs in various dimensions when solving this problem. In Java, as long as we leverage inheritance and dynamic dispatch, we can get close, although you should be aware that the approach is still unsatisfying in the ways we have discussed.

14.9 Case Study: Exceptions

Previously, we discussed how to *raise exceptions* using the throw statement:

```
throw new IllegalArgumentException();
```

The subject of the `throw` statement is an expression that evaluates to an exception object. This led to the question of whether we could define our own exceptions. Indeed we can via subclassing; exceptions turn out to be a great—if not controversial—example of inheritance in Java.

14.9.1 Defining Exceptions

In Java, we define our own exception by writing a subclass for the `Throwable` class or one of its subclasses. The `Throwable` class hierarchy looks like this:

```
                Throwable
                    |
            -----------------
            |               |
        Exception         Error
            |               |
    -----------------      ...
    |               |      (3)
   ...       RuntimeException
   (1)              |
                   ...
                   (2)
```

You can create subclasses for your exception at any of the three points indicated above. Each point represents one of three possible types of exceptions in Java:

1. *Checked exceptions* are subclasses of `Exception` (or its subclasses—technically any subclass of `Throwable` that is not also a subclass of `Error` or `RuntimeException`). Checked exceptions are exceptional conditions that a program ought to anticipate and recover from. For example, the `Scanner` class's constructor throws `FileNotFoundException` if the given file does not exist on disk. This is an exceptional condition, but one that is easily recoverable from—inform the user and exit or ask the user for another file.
2. *Unchecked exceptions* or runtime exceptions are subclasses of `RuntimeException`. These exceptions arise from conditions internal to the application that are not recoverable from. An example of an unchecked exception is the `IllegalArgumentException` that we've thrown when a pre-condition on a method is violated. In such a scenario, we don't want to try to recover from this error as the condition should never occur.
3. *Errors* are subclass of the `Error` class. They are similar to unchecked exceptions in that they are situations in which the program cannot recover, but the conditions in which they occur are external to the application, rather than internal. For example, if we run out of memory—a situation that is not a problem with our program, *per se*, and is not recoverable—Java will throw a `VirtualMachineError`.

In many cases, the exceptions provided by the Java standard library are sufficient. For example, the descendants of `RuntimeException` such as `IllegalArgumentException`, `NullPointerException`, and `IllegalStateException`, cover a lot of the common cases of pre-condition and invariant violation that we might encounter in our code. However, if we would like to provide a more specific exception, *e.g.*, a `BankAccountNegativeException` for when the invariant that our bank account balance goes negative, we can create a subclass of the appropriate type.

14.9.2 Handling Exceptions

Checked exceptions require that we recover from them in some way. Java provides two mechanisms for this:

1. The `throws` clause on methods indicating that a method generates an exception and that any caller of this method must handle it. For example, when we created a new `Scanner` with a File, we had to add `throws FileNotFoundException` to indicate that the method that contained this code could throw this exception.
2. Try-catch statements which allow us to *try* to execute some code that may throw an exception and then *catch* any exceptions that arise from them.

In Java, all *checked exceptions* must be dealt handled in one of these two ways—throws clauses or try-catch. While you can use these mechanisms to document and catch runtime exceptions and errors, it is strongly advised not to do so—these exceptions occur specifically because you cannot recover from them.

The try-catch has the following form, *e.g.*, to properly create a `Scanner` from a `File`:

```
import java.io.FileNotFoundException;
// ...
Scanner in = null;
File file = new File("foo.txt");
if (file.exists() && !file.isDirectory()) {
    try {
        in = new Scanner(new File("foo.txt"));
    } catch (FileNotFoundException e) {
        System.out.println("Error: file not found " + e.getMessage());
    }
    if (in != null) {
        // Use the scanner...
    }
}
```

Here, we try to create a scanner that reads from `foo.txt`. If no exception is thrown, then `in` is loaded with an appropriate `Scanner` object. If the line `in = new Scanner(new File("foo.txt"))` throws a `FileNotFoundException`, then control flows to the *exception handler* for the `FileNotFoundException`, *i.e.*, the block of code that prints the error message. In either case, control flows to the line after the try-catch block—the if-statement. We check to see if `in` is `null` after the try-catch block because if `FileNotFoundException` is thrown, then `in` will never be loaded with a `Scanner` object.

Note that even though we check to see if the file exists, we still need to use a try-catch to catch the potential `FileNotFoundException`. This is because `FileNotFoundException` is a checked exception, but it is marked as such because even though we have done the proper check, the file might be deleted from disk or made unavailable after the check executes! Also note that we cannot declare the `Scanner` variable inside the try-catch block. This is because of a technicality with scoping. If we wrote this code:

```
import java.io.FileNotFoundException;
// ...
if (file.exists() && !file.isDirectory()) {
    try {
        Scanner in = null;
        File file = new File("foo.txt");
        in = new Scanner(new File("foo.txt"));
    } catch (FileNotFoundException e) {
        System.out.println("Error: file not found " + e.getMessage());
    }
    // Error: `in` not in scope!
    if (in != null) {
        // Use the scanner...
    }
}
```

We receive a compiler error stating that the occurrence of `in` in the conditional is not in scope. This is because the lifetime of a local variable is the curly braces that enclose it. Therefore, the lifetime of `in` in this case is the try-block. We want `in` to exist outside of the try-block, so we have to declare it outside of the try-block.

We can catch any number of exceptions in a try-catch block by nesting branches, *e.g.*:

```
try {
    // Code that throws either FileNotFoundException
    // or IndexOutOfBoundsException...
} catch (FileNotFoundException e) {
    // If FileNotFoundException is thrown...
} catch (IndexOutOfBoundsException e) {
    // If IndexOutOfBoundsException is thrown...
}
```

Furthermore, a catch block catches a specified exception as well as any of its subtypes. `FileNotFoundException` is a subtype of `IOException`, so we could use `IOException` in its place if we also anticipated other subtypes of `IOExceptions` to be thrown or an `IOException` itself (usually reserved for general IO problems):

```
try {
    // Code that throws any IOException...
} catch (IOException e) {
    // Catches any IOException or a subclass of IOException.
}
```

Because of subtyping, we could theoretically catch any exception—checked or unchecked—by catching `Exception`. This is typically used as a "catch-all" catch case:

```
try {
    // Code that can throw an exception
} catch (Exception e) {
    // Handles any (checked) exception...
}
```

However, this is usually undesirable because if you are required to handle a particular (checked) exception, you will likely want to handle it in its own specific way, *e.g.*, issuing an error to the user. Using this "catch-all" style does not allows this. Worse yet, is combining "catch-all" with *swallowing the exception*:

```
try {
    // Code that can throw an exception
} catch (Exception e) {
    // Do nothing if we receive an exception
}
```

This silently handles any checked exception that is raised by the code in the try-block. While convenient, again, you will likely need to have some recovery logic for each specified checked exception that is raised. At the very least, you will want to avoid swallowing the unchecked exceptions that this code could throw as these represent unrecoverable errors.