3 Additional Java Features

This chapter discusses more Java features, equipping the readers with the tools to write clear, effective, and maintainable code in Java. Because Java programs revolve around object-oriented programming, we will look at writing Java classes in more detail. Most often, code needs to refer to an instance of another class or the same class. This can be accomplished through the concept of references, which we discuss after that. Understanding references helps us better understand how parameters are passed in Java. This is discussed next. Following that, we touch on two important concepts: inheritance (deriving a new class that acquires the qualities of an existing one) and interfaces (specifying what a class must implement).

3.1 Class Mechanics

Since all activities in a Java program occur within a class, we have been using classes without saying much about them. In general, the concept of classes in Java and in Python are similar. A *class* is a template that defines the form of an object. It specifies both the data and the code that will operate on that data. Both Python and Java use a class specification to construct objects. Therefore, *objects* are instances of a class—and a class, to put it another way, is essentially a set of plans that specify how to build an object. Hence, a class is a logical abstraction. It is not until an object of that class has been created that a physical representation of that class exists in memory.

Members of a class include methods and variables (also known as *instance variables*). It helps to think of a(n ordinary) method as a function that lives as part of that class and can refer to instance variables.

The Anatomy of a Class

When we define a class, we generally specify two things: instance variables and methods. Some classes might contain only instance variables or methods. A typical class looks as follows:

```
class Classname {
    // declare instance variables
    type var1;
    type var2;
```

```
    // ...
    type varN;

    // declare methods
    type method1(parameters) {
        // body of method
    }
    type method2(parameters) {
        // body of method
    }
    // ...
    type methodN(parameters) {
    // body of method
    }
}
```

Tips

If we could get a little ahead of ourselves, a well-designed class should define one and only one logical entity. For example, a class that contains student information should not contain information about stock market, etc.

It is worth noting that up to this point, each class that we wrote has a `main` static method. In general, a class does not require a `main` method. It is needed only when the class is the starting point of our program. Indeed, a program can have multiple classes.

Defining a Class

To illustrate how classes are defined and work, we will develop a class that keeps information about vehicles, such as cars, vans, and trucks. This class is called `Vehicle`, and it will store three pieces of information about a vehicle: the number of passengers that it can carry, its fuel capacity, and its average fuel consumption (in miles per gallon).

The first version of `Vehicle` is shown next. It defines three instance variables: `passengers`, `fuelCap`, and `mpg`. Notice that `Vehicle` does not contain any methods. Thus, it is currently a data-only class. We will add methods to it later.

```
public class Vehicle {
    int passengers; // number of passengers
    int fuelCap;    // fuel capacity in gallons
    int mpg;        // fuel consumption in miles per gallon
}
```

This public class has to be saved inside `Vehicle.java`. Remember that the name of a public class has to match the name of the file storing it. Now, to actually create a `Vehicle` object, we will write `new Vehicle()`, like so:

```
// create a Vehicle object called minivan
Vehicle minivan = new Vehicle();
```

After this statement executes, `minivan` will be an instance of `Vehicle`. Each time we create an instance of a class, we are creating an object that contains *its own copy of each instance variable* defined by the class (the "blueprint"). Thus, every `Vehicle` object will contain its own copies of `passengers`, `fuelCap`, and `mpg`. To access these variables, we will use the dot (`.`) operator. The dot operator links the name of an object with the name of a member. The general form of the dot operator is shown here:

```
objectName.memberName
```

For example, we would write

```
minivan.fuelCap = 16;
```

which sets the instance variable `fuelCap` inside `minivan` to 16.

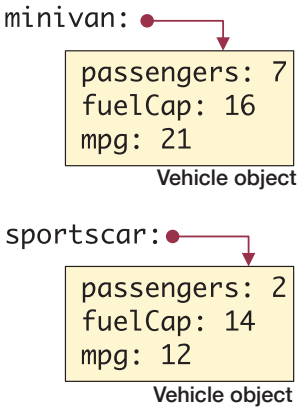
For a complete example, the code below shows how the `Vehicle` class is used from another driver class `VehicleDemo`.

```
1  public class VehicleDemo {
2      public static void main(String args[]) {
3          Vehicle minivan = new Vehicle();
4          int range;
5          // assign values to fields in minivan
6          minivan.passengers = 7;
7          minivan.fuelCap = 16;
8          minivan.mpg = 21;
9          // compute the range assuming a full tank of gas
10         range = minivan.fuelCap * minivan.mpg;
11         System.out.println(
12             "Minivan can carry " + minivan.passengers +
13             " with a range of " + range);
14     }
15 }
```

When we compile the two classes, we will see two `.class` files: `Vehicle.class` and `VehicleDemo.class`. To run it, we will need to run with the class that has a `main` method, which is `VehicleDemo` in this case:

```
java VehicleDemo
```

Can we create two separate instances of the same class? Of course, we can—that is how classes are intended to work. In Code 3.1, the class `TwoVehicles` demonstrates this. We will notice that each `Vehicle` object has its own copy of the variables. As you can see, `minivan`'s data is completely separate from the data contained in `sportscar`.



Two instances of the `Vehicle` class have their own copies of the instance variables.

Method Overloading

In Java, two or more methods within the same class can share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be *overloaded*, and the process is referred to as *method overloading*.

Code 3.1: Example of two instances of the same class.

```
1  public class TwoVehicles {
2      public static void main(String args[]) {
3          Vehicle minivan = new Vehicle();
4          Vehicle sportscar = new Vehicle();
5          int range1, range2;
6          // assign values to fields in minivan
7          minivan.passengers = 7;
8          minivan.fuelCap = 16;
9          minivan.mpg = 21;
10         // assign values to fields in sportscar
11         sportscar.passengers = 2;
12         sportscar.fuelCap = 14;
13         sportscar.mpg = 12;
14         // compute the ranges assuming a full tank of gas
15         range1 = minivan.fuelCap * minivan.mpg;
16         range2 = sportscar.fuelCap * sportscar.mpg;
17         System.out.println(
18             "Minivan can carry " + minivan.passengers +
19             " with a range of " + range1);
20         System.out.println(
21             "Sportscar can carry " + sportscar.passengers +
22             " with a range of " + range2);
23     }
24 }
```

Method overloading is one of the ways that Java implements polymorphism. In general, to overload a method, simply declare different versions of it. The compiler takes care of the rest. But we must observe one important restriction:

> The type and/or number of the parameters of each overloaded method must differ.

To be able to discuss this important restriction more precisely, we have to understand the concept of a method signature.

Method Signature

Two components of a method declaration comprise the *method signature*—the method's name and the parameter types. Therefore, the methods

```
public double calculateAnswer(double wingSpan, int numberOfEngines,
                              double length, double grossTons) {
    //do the calculation here
}
public int fooBar(int foo, double bar) {
          //compute fooBar here
}
```

have signatures (respectively):

```
calculateAnswer(double, int, double, double)
fooBar(int, double)
```

It is important to note that parameter variable names and the method's return type are *not* part of the signature.

Back to Overloading

More precisely, if two methods have the same name, Java distinguishes between methods using method signatures. Therefore, it is okay to overload names as long as they have different signatures. For example, we can implement a suite of method called `draw` for displaying different kinds of data:

```
public void draw(String s) { ... }
public void draw(int i) { ... }
public void draw(double f) { ... }
public void draw(int i, double f) { ... }
```

These are all different methods because they all have different signatures. When we call `draw`, the method with the matching signature will be used. As a concrete example:

```
draw("Hello"); // will use draw(String)
draw(24); // will use draw(int)
draw(25, "H"); // will cause an error because nothing matches (int, String)
```

Constructor Overloading

Constructors can also be overloaded, allowing us to construct objects in a variety of ways. As an example, consider the program below.

Notice that we use `this.x` to refer to the member variable `x` of that particular object. In the present case, because `x` is not ambiguous, we can shorten `this.x` to simply `x`.

```
class MyClass {
    int x;
    MyClass() {
        System.out.println("Inside MyClass().");
        this.x = 0;
    }
    MyClass(int i) {
        System.out.println("Inside MyClass(int).");
        this.x = i;
    }
    MyClass(double d) {
        System.out.println("Inside MyClass(double).");
        this.x = (int) d;
    }
    MyClass(int i, int j) {
        System.out.println("Inside MyClass(int, int).");
        this.x = i * j;
    }
}
```

```
public class OverloadConsDemo {
    public static void main(String args[]) {
        MyClass t1 = new MyClass();
        MyClass t2 = new MyClass(88);
        MyClass t3 = new MyClass(17.23);
        MyClass t4 = new MyClass(2, 4);
        System.out.println("t1.x: " + t1.x);
        System.out.println("t2.x: " + t2.x);
        System.out.println("t3.x: " + t3.x);
        System.out.println("t4.x: " + t4.x);
    }
}
```

The code shows how a class can contain multiple constructors of different signatures—constructor overloading. Readers are encouraged to run this code to see how it works in action.

The Static Business: Methods and Variables

There will be times when we will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed through an object of its class, but it is possible to create a member that can be used by itself, without reference to a specific object instance.

To create such a member, precede its declaration with the keyword `static`. When a member is declared `static`, it can be accessed even before any objects of its class are created, and without reference to any object. Both methods and variables can be declared to be static.

The most common example of a static member is `main`. The `main` method is made static because it is called directly by the JVM to begin our program.

Outside the class that defines it, to use a static member, we only need to specify the name of its class followed by the dot operator. No object needs to be created. For example, consider an implementation of the class `Timer`:

```
class Timer {
    static int count;
    double instVar1;
    // other methods omitted
}
```

Among other things, `Timer` has a static variable called `count` and an instance variable `instVar1`. Because one is static and one is not, `count` and `instVar1` are accessed differently.

- Being a static variable, `count` is part of the `Timer` class, so the line `Timer.count = 10` will just work. However, all objects (i.e., instances) of the class `Timer` share the same `count`.
- Being an instance variable, `instVar1` belongs to a particular object of the `Timer` class. This means: `Timer.instVar1` makes no sense because Java has no idea which object we are talking about. Furthermore, if `a` and `b` are different objects of the `Timer` class, `a.instVar1` and `b.instVar1` are different variables (they share nothing).

From this perspective, variables declared as static are, in a sense, global variables. When an object is declared, no copy of a static variable is made. Instead, all instances of the class share the same static variable.

Below we summarize the key differences between static and non-static (instance) methods:

- Static methods are called through its class name, without any object of that class being created. We have seen an example of this already: the `sqrt` method, which is a static method within Java's standard Math class. For this reason:

  > Static methods/variables are properties of a class, not a specific object whereas instance methods/variables are called with respect to an object.

- An important implication of this is that static methods can't access "my" instance variables (specifically there is no concept of **`this`**), because there is no notion of "me."

Before we move on, we may wonder why static methods make sense? For example, some classes like `Math` is never meant to be instantiated—why would anyone have to instantiate `Math` to invoke `sqrt` or `round`?

Inside a class, constructors and (ordinary) methods can refer to its own object using the special "variable" called **`this`**. Therefore, for example, we write **`this`**`.foo` to reference the member variable `foo`. When there is no ambiguity, writing just the variable name (e.g., `foo`) is sufficient.

3.2 References

References are the general concept of pointing to another "thing," quite akin to the idea of pointers that refer to another location/object. We will now explore the concept of references in Java, beginning with a quick puzzle.

Puzzle: The Mystery of Felines

To get started, we will work through a puzzle using our intuition, plus what we have seen from other programming languages.

First, we will define a class called `Feline`. The only important bit here is the instance variable `w`, so everything else is omitted.

```
class Feline {
    int w;
    // other detail omitted
}
```

We will now look at the following code that uses this class.

```
Feline a = new Feline();
a.w = 5;   // sets the value of w inside a to 5
Feline b = a;
b.w = 9;
```

Can we predict the values of `a.w` and `b.w` at the end of the execution?

Let us then take a look at a nearly-identical piece of code that works on the type **`int`**:

```
int x = 5; // declares a new variable and set it to 42
int y = x;

y = 9;
```

Spoiler Alert. When we run these programs, we will find the following values:

a.w	9
b.w	9
x	5
x	9

What are the values of `x` and `y` at this point? Readers may want to try them out first before reading further.

We quickly learn that while changing a value inside `b` affects `a`, changing `y` does *not* affect `x`. But why? To understand this, there are *three* ideas involved:

1. Everything we store is a sequence of bits (encoded appropriately).
2. Primitive vs. Reference types.
3. The Rule of Equal (RoE)

Everything Is A Sequence Of Bits

As we probably have seen, information is stored in memory as a sequence of `1`s and `0`s. For example, we know

- the number 65 is most often kept as `01000001`
- the number 3.1415 is kept as `01000000010010010000111001010110`
- the letter A is most often kept as `01000001`, which coincidentally (or not) is the same as 65 above.

This means if we peek at an arbitrary location in the physical memory, all we see is a sequence of `1`s and `0`s. We have no idea whether we're looking at a (whole) number, a floating-point number, or a string symbol. Fortunately, using internal bookkeeping and other mechanisms, Java knows how to make sense of these. Clearly, there is more to this story—details, however, are beyond the scope of this book.

Primitive vs. Reference Types

Java distinguishes between two kinds of types because they are stored differently. There are eight primitive types in Java:

`boolean`, `byte`, `char`, `short`, `int`, `long`, `float`, `double`

Everything else is a reference type, including arrays of any kind. If we do not feel like memorizing all the names, it helps to know that every lower-case type is primitive.

Now why do we distinguish between them? When we declare a variable of a certain type, here is roughly what happens:

- Java sets aside enough bits in the memory to hold data of that type. For example, if we declare an `int`, it will set aside a box for 32 bits. Declaring a `double` will set aside 64 bits.
- It remembers how the variable name we choose maps to that location.

Assigning a value to such a variable writes into the corresponding space. For example, `x = 65` will write `01000001` into the box for `x`. Notice that Java does not permit reading from a box that the program has not written to. This is for good reasons: no useful data is there in that physical memory location.

This behavior applies to all primitive types: for primitive types, the actual data is kept in that box.

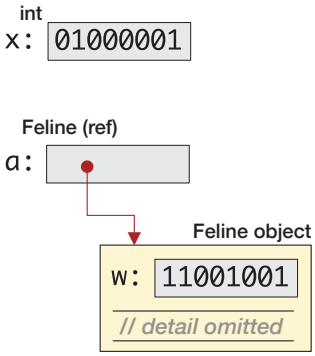
For reference types, the story is different. Because reference types tend to be larger and hence clumsier to move around, the box we have been talking about stores a pointer (known as a reference) to where the actual data resides. In greater detail, when we write

```
Feline a = new Feline();
```

the following happens approximately:

- Because of the expression `new Feline()`, Java sets aside enough space for an instance of `Feline`, enough to store every instance variable and some more—and the constructor is called to fill the instance variables and does the setting up.
- The statement `Feline a = ...` causes Java to set aside enough bits to hold a `Feline` reference. Then, these bits are assigned the reference (i.e., pointer) to the object we have just created. Java also remembers that the name `a` refers to this "box" of bits.

Hence, the box labeled `a` doesn't really contain the object but a pointer (aka. reference) to where the object lives.



As a primitive-type variable, `x` (of type `int`) stores the actual data directly in the space allocated for the variable. As a reference-type variable, `a` (of type `Feline`) stores the reference to where the actual object is.

Tips

In short, remember that

- for primitive types, the variables store the actual data; but
- for reference types, the variables store the location of the actual data (storing pointers/references).

The Rule of Equals (RoE)

This rule is concerned with assignment statements. The assignment statement `y = x` says

> Copy all the bits from `x` into `y`.

In effect, this means:

- If `x` and `y` are a primitive type (such as an `int`), the actual data (i.e., the actual bits encoding that integer number) is copied.
- However, if `x` and `y` are a reference type (such as a `Feline`), the reference (i.e., the bits storing the location) is copied.

3.3 Extension: Parameter Passing

Now that we have discussed the rule of equals, it is easy to describe how parameters are passed in Java. The principle is as follows:

> Parameter passing copies the bits in the same way as the rule of equals. That is, when a parameter to a method is passed, Java simply copies the bits into the new scope.

To understand this principle in context, consider the following example:

Example. As defined below, the method `update` changes the contents of its object parameter `f` and a primitive-typed parameter `x`.

```
void update(Feline f, int x) {
    f.w = f.w + 42;
    x = x + 42;
}
```

If we write the following code, can we predict what will happen to `a` and `x`?

```
Feline a=new Feline();
x = 5;
update(a, x);
// a.w = ? x = ?
```

Because `a` has a reference type, when `a` is copied into `f`, Java copies the location. Hence, the variable `f` inside `update` refers to the same object as `a`, so modifying `f.w` leads to changes seen via `a`. This means, `a.w` has become 47. Differently, because `x` has a primitive type, the actual value of `x` is copied, so changing `x` inside `update` leaves the original `x` unaffected.

3.4 Inheritance

Inheritance is a mechanism in which one class acquires the qualities of another class. More specifically, it permits programmers to begin writing a class by acquiring the qualities of another class—known as the parent class—and further refine it appropriately. This is a central concept in object-oriented programming and is widely supported in languages with object-oriented programming features. It promotes and supports the concept of hierarchical classification. With inheritance, a class only needs to define the unique qualities that make it differ from the parent class. In other words, it inherits all the characteristics of the parent and is refined with its own unique qualities. Without inheritance, each class would have to explicitly define all of its characteristics all over again, from scratch.

What exactly do we mean by acquiring the qualities of the parent class? We will illustrate this, as well as the mechanics of inheritance in Java, by way of example. Below is a (contrived) example of a simple class called `Animal`:

```
class Animal {
   void walk() { System.out.println("Animal: I'm walking."); }
   void eat()  { System.out.println("Animal: I'm eating."); }
}
```

The class has two methods `walk` and `eat`. Next, we are going to build a `Pet` class, which will mechanically inherit from `Animal`—that is, it has all the services/properties/characteristics of the `Animal` class, and is further refined by the code we are writing for `Pet`.

```
class Pet extends Animal {
   void play(String with) {
```

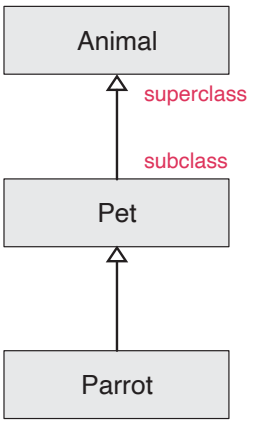
```
        System.out.println("Pet: I enjoy playing with " + with);
    }
}
```

The keyword `extends`. The keyword here is `extends`. In this case, the class `Pet` extends `Animal`. Notice that a `Pet` object can `walk`, `eat`, and—now additionally—`play`.

Multilevel Inheritance. Inheritance can be applied again. For example, the `Parrot` class extends from `Pet`, which in turn, extends from `Animal`.

```
class Parrot extends Pet {
    void sing() { System.out.println("Parrot: I'm singing"); }
}
```

Notice that `Parrot` has all the "abilities" of the parent class (`Pet`), which, in turn, has all the "abilities" of its parent. In the terminology of object-oriented programming, we say that `Animal`is a *superclass* of `Pet`. In the opposite direction, we have `Pet` is a *subclass* of `Animal`. As another example, `Parrot` is a subclass of `Pet` and `Pet` is a superclass of `Parrot`.



As an example, `Animal` is a superclass of `Pet`, which is a superclass of `Parrot`. This means, `Parrot` is a subclass of `Pet`, which is a subclass of `Animal`.

Observations and Remarks. We use the keyword `this` to refer to the instance of the present class. For example, write `this.member` to refer to the instance variable `member` of this instance. We use the keyword `super` to refer to the parent object. It is important to note that what is private to the superclass is inaccessible from the subclass. This means that if `x` is a private instance variable of a parent class, `x` will not be accessible in the subclass.

But then how is an object with inheritance constructed? The basic idea is simple: the constructor of the subclass constructs that class, and the constructor of the superclass constructs the portion relevant to the superclass. This dictates that the superclass's constructor essentially has to run prior to the subclass's. Explicitly, the subclass's constructor can call the superclass's constructor using the syntax: `super(...)`, where `...` indicates the parameters one wishes to pass to that constructor. However, if no superclass's constructor is explicitly called, Java calls the default constructor as if the line `super()` is inserted at the top of the subclass's constructor.

3.5 Interfaces

We now turn our attention to another construct in Java that allows us to express what a class must do (specification) separately from how it will be done (implementation). A great motivating factor is when writing code in a large software system, where one programmer might depend on the code that another programmer has yet to write. The concept of an interface offers a means to promise that a class that satisfies—or implements, in a technical term—such an interface will have methods `a`, `b`, `c`, and so on. In a sense, it offers a way to write and use a contract:

- From the point of view of providers, there can be many classes that implement a contract.
- From the point of view of users, any implementation can be used, for the most part, as long it implements the interface.

A software construction course will delve much deeper into the detail. In this book, we will only touch on this briefly.

The Anatomy of an Interface

To define an interface, use the `interface` keyword. For example,

```
public interface Series { ... }
```

defines an interface called `Series`. The optional modifier keyword `public` works similarly to that of classes, making it publicly visible.

Inside an interface, we can specify (i) methods that are needed to satisfy the contract and (ii) variables that are part of the contract. Methods declared as part of an interface are implicitly public. Variables are implicitly public, static, and final—and they must be initialized. The rest of this section will focus on using interfaces for methods.

A typical interface looks as follows:

```
interface InterfaceName {
    //declare methods (made to be public by default)
    type method1(parameters);
    type method2(parameters);
    // ...
    type methodN(parameters);
}
```

Defining an Interface

We will learn the mechanics of Java interfaces by way of example. To begin, we will write an interface for classes that generates a series of numbers. One way to describe an infinite series is to keep asking for the next number. The following interface promises the ability to retrieve the next number in the series (via `next`) and to rewind back to the start (via `reset`). Because the public interface is called `Series`, it has to live in a file named `Series.java`:

Code 3.2: An example interface called `Series`.

```
1 public interface Series {
2     int next();    // return the next number
3     void reset();  // restart
4 }
```

As written, this promises two public methods: `next` and `reset`. It is not possible to promise any form of constructors. Notice also that defining an interface simply creates a contract. It does not yield an implementation that satisfies it. That implementation will come in the form of a class. We will look at this next.

Implementing and Using an Interface

To implement an interface, we define a class and indicate that the class implements this interface. The class below represents the series $0, 2, 4, 6, \ldots$.

```
class ByTwo implements Series {
    private int val;

    public ByTwo() { val = 0; }

    public int next() {
        int prevVal = val;
        val += 2;
        return prevVal;
    }
    public int magic() { return 42; }
    public void reset() { val = 0; }
}
```

How do we use this class? First, we can still use it normally. For instance, the following code, as usual, instantiates the class and calls its method:

```
ByTwo seq = new ByTwo();
System.out.println(seq.next());
System.out.println(seq.magic());
```

Notice that `magic` is not promised in the `Series` interface and can be accessed as usual.

Next, from the point of view of types, because `ByTwo` implements `Series`, every `ByTwo` object is also a `Series`. Hence, the following code is possible:

```
Series altSeq = new ByTwo();
System.out.println(altSeq.next());
```

Being a `Series`, the variable `altSeq` has access to everything that the interface promises. However, with this declaration, it is not possible to access the `magic` method via `altSeq`—the method simply does not exist from the point of view of a `Series`.

Observations and Remarks. There can be multiple classes that implement an interface. We wrote `ByTwo` earlier. To demonstrate this concept, we will write another class that implements this interface:

```
class ConstFive implements Series {
    public int next() { return 5; }
    public void reset() { }  // reset does nothing.
}
```

So far, a class implements one interface. But in fact, a class can implement many interfaces. In terms of syntax, we write

```
class A implements I1, I2, I3 { ... }
```

and this would mean class `A` promises that it satisfies interfaces `I1`, `I2`, and `I3`. Finally, because there is no implementation—merely a specification/contract—we cannot instantiate (`new`) an interface.

Interfaces in the Wild

Suppose for a moment that we wish to add up the first `n` number from the `ByTwo` class. The code below is typical:

```
int sumN(ByTwo seq, int n) {
    seq.reset();
    int sum=0;
    for (int i=0;i<n;i++) { sum += seq.next(); }
    return sum;
}
```

How would one write code for the same problem for `ConstFive`? We would essentially write the same code. However, we can avoid repeating ourselves by remembering that both `ByTwo` and `ConstFive` are a `Series`. Hence, it is possible to take as input a `Series` and that would work with any class that satisfies the `Series` interface, like so:

```
int sumN(Series seq, int n) {
    seq.reset();
    int sum=0;
    for (int i=0;i<n;i++) { sum += seq.next(); }
    return sum;
}
```

This code will work with `ByTwo` and `ConstFive`–and any `Series` we might come across in the future.

Exercises

Exercise 3.1. Write a public class `Counter` to represent a counter value with the following specification:

- The only constructor is public and takes no input. It internally sets the counter to 0.
- The class has a public method `void increment()` that increments the counter by 1.
- The class has a public method `void decrement()` that decrements the counter by 1. If the counter goes below 0 after decrementing, reset the counter to 0.
- The class has a public method `int getValue()` that returns the current value of the counter.

Exercise 3.2. Extend the `Counter` class from Exercise 3.1. as follows:

- Add a form of `increment` where it takes an integer indicating how much to increment by, i.e., `void increment(int delta)`.
- Add a form of `decrement` where it takes an integer indicating how much to decrement by, i.e., `void decrement(int delta)`.

Make sure that the counter value doesn't go below 0.

Exercise 3.3. As discussed earlier in the chapter, a class can have multiple constructors. Add the following constructure to the `Counter` class from Exercise 3.2.. The new constructor will take in an integer (`int`), which will be the starting value of the counter.

Exercise 3.4. Suppose we write the following code snippet:

```
class Foo { int a; }
void update(double x, Integer y, Foo z, Foo t) {
    x += 1; y += 2; z.a = 3; t = null;
}
```

Predict without running what will be printed if we run the following code:

```
double  p = 1.0;
Integer q = 2;
Foo     r = new Foo(); // note: r.a defaults to 0
Foo     s = new Foo(); // note: s.a defaults to 0

update(p, q, r, s);
System.out.println(p + " " + q + " " r + " " + s);
```

Exercise 3.5. We wrote an interface `Series` in Code 3.2. Implement a class `Geometric` that represents the series: 1, a, a^2, a^3, ..., where a is an integer parameter taken in via the constructor. More specifically, the class has a constructor

```
public Geometric(int a) { ... }
```

that takes as input the parameter `a`. Additionally, the class implements the `Series` interface and returns the series described earlier.

Exercise 3.6. Write a public class `Circle` to represent a circle with the following specification:

- The only constructor is public and takes as input the radius of this circle of type `double`.
- The class has a public method `void setRadius(double r)` that sets the radius of this circle.
- The class has a public method `double getRadius()` that returns the radius of this circle.
- The class has a public method `double getArea()` that returns the area of this circle. Remember that the area of a circle with radius r is $\pi \cdot r^2$.

Exercise 3.7. Write a public interface `GeometricShape` that promises the following methods:

```
// returns the area of this object
double getArea();

// returns the length of the perimeter
double getPerimeter();
```

Exercise 3.8. Update the `Circle` class from Exercise 3.6. so that it implements the interface `GeometricShape` from Exercise 3.7.

Exercise 3.9. Consider the following `Bird` class:

```
class Bird {
    public String name;
    public Bird(String name) { this.name = name; }
    public void speak() { System.out.println("Tweeet"); }
}
```

Write a class `Parrot` that extends from `Bird` with the following characteristics:

- There is a constructor that accepts a `String` for name. You will need to call the constructor of the super class using the `super` keyword with the appropriate parameters.
- Override the method `speak` so that it prints out "Hello, World" instead of the previously specified text.

Exercise 3.10. Implement the following two public classes:

- `Square` represents a square, where the public constructor `Square(int l)` takes a integer side-length parameter.
- `Rectangle` represents a rectangle, where the public constructor `Rectangle(int w, int h)` takes integer side-length parameters.

Both classes will implement the `GeometricShape` interface from Exercise 3.7.

How can we make an array that can store any combination of `Circle`, `Square`, and `Rectangle`? Ultimately, we will write a method

```
double averageArea(.... shapes)
```

that takes as input an array of these geometric shapes (the exact type is left for you to decide) and returns the average area of the shapes.

Chapter Notes

This chapter is only scratching the surface of Java programming and object-oriented design. The goal of a chapter like this in a Data Structures book is to introduce enough Java concepts to appreciate the core material—the design and implementation of data structures and algorithms. However, Java is deep and sophisticated, connected to the fascinating software-engineering principles. Excellent books by Herbert Schildt—*Java: A Beginner's Guide* [Sch18a] and *Java: The Complete Reference* [Sch18b]—inspired and informed the style and contents of this chapter. Readers are encouraged to read these two volumes, as well the beginning *Core Java* book [Hor16], for further elements of Java. For a manual on writing effective Java code, Joshua Bloch's *Effective Java* [Blo08] is a real gem.