8.17 Wrap-Up

In this chapter, we presented additional class concepts. The `Time` class case study showed a complete class declaration consisting of `private` data, overloaded `public` constructors for initialization flexibility, *set* and *get* methods for manipulating the class's data, and methods that returned `String` representations of a `Time` object in two different formats. You also learned that every class can declare a `toString` method that returns a `String` representation of an object of the class and that method `toString` can be called implicitly whenever an object of a class appears in the code where a `String` is expected. We showed how to `throw` an exception to indicate that a problem has occurred.

You learned that the `this` reference is used implicitly in a class's instance methods to access the class's instance variables and other instance methods. You also saw explicit uses of the `this` reference to access the class's members (including shadowed fields) and how to use keyword `this` in a constructor to call another constructor of the class.

We discussed the differences between default constructors provided by the compiler and no-argument constructors provided by the programmer. You learned that a class can have references to objects of other classes as members—a concept known as composition. You learned more about `enum` types

and how they can be used to create a set of constants for use in a program. You learned about Java's garbage-collection capability and how it (unpredictably) reclaims the memory of objects that are no longer used. The chapter explained the motivation for `static` fields in a class and demonstrated how to declare and use `static` fields and methods in your own classes. You also learned how to declare and initialize `final` variables.

You learned that fields declared without an access modifier are given package access by default. You saw the relationship between classes in the same package that allows each class in a package to access the package-access members of other classes in the package. Finally, we demonstrated how to use class `BigDecimal` to perform precise monetary calculations.

In the next chapter, you'll learn about an important aspect of object-oriented programming in Java—inheritance. You'll see that all classes in Java are related by inheritance, directly or indirectly, to the class called `Object`. You'll also begin to understand how the relationships between classes enable you to build more powerful apps.