Data & Procedure Java review

The primary purpose of the studies in this book is to understand *efficient computation*, but throughout we study this via examples written using the Java programming language. Thus, to understand the material, you need some Java mastery. While you may already have some familiarity with the basic concepts, often students don't quite fully understand the object-oriented concepts on which Java is based. Because of these concepts' importance to our study, you need to understand both the terminology and the concepts of object-oriented programming.

This chapter is a review of object-oriented programming. Because we're only reviewing what you should already know, this chapter skates quickly over many topics. Don't let this mislead you into thinking that the details are unimportant: You need to understand all of it to be successful in fully understanding the rest of the book. If you have any questions about any of it, you should either consult a general Java reference or ask your instructor for further clarification.

9.1. Objects and classes

A Java program is written as a set of *classes*; as the program runs, it will create *instances*, or *objects*, of these classes.

For example, we may want to write a program so that library patrons can locate books in a library. Since the program is likely to be fairly large, it would probably include several classes. Among these classes we would include a class named *Book*; when the program is run, a number of Book instances would be created, each object corresponding a real library book. Thus, there is just one Book class, but potentially a program could have many Book instances at any point in time.

Classes can contain many elements. The two most important categories of elements are *instance variables* and *instance methods*. An *instance variable* is a property of an instance of that class: I like to think of each instance variable as something that the book remembers. An *instance method* is a request or command that can be given to the instance.

For our Book class, two important attributes of a book are its author and title; these two attributes become instance variables. We might want to ask a book who its author is, what its title is, and whether it is a duplicate of another book; each of these possible actions translates to an instance method.

```
class Book {
    String author;
    String title;

    String getAuthor() { return author; }
    String getTitle()  { return title;  }

    boolean isCopy(Book query) {
        return author.equals(query.author) &amp;& title.equals(query.title);
    }
}
```

With the Book class defined as above, we might legally write the following in a method of another class.

```
Book theOdyssey;
Book myBook;
// other code omitted, including initializing the variables
// to refer to Book objects.
System.out.println(myBook.getTitle() + ", by " + myBook.getAuthor());
System.out.println("  " + myBook.isCopy(theOdyssey));
```

Our Book definition, though, is missing a crucial element to make it useful: It provides no facility for creating Book instances. (The previous code fragment illustrating the usage of the Book class skipped over this with a comment saying that code was omitted.) To facilitate instance creation, a class needs *constructors*, each defining what to assign as initial values to the instance variables when a new instance is created. (Technically, if you do not define any constructors, Java will define one for you, taking no arguments and doing nothing. Generally, your programs should not depend on this, though.) Such a definition is in Figure 9.1.

Figure 9.1: Defining the `Book` class: Take two.

```
class Book {
    String author;
    String title;

    Book(String inAuthor, String inTitle) {
        author = inAuthor;
        title = inTitle;
    }
```

```
    Book(Book original) { // creates copy of original book
        author = original.author;
        title = original.title;
    }

    String getAuthor() { return author; }
    String getTitle()  { return title;  }</p>

    boolean isCopy(Book query) {
        return author.equals(query.author) && title.equals(query.title);
    }
}
```

Constructor definitions look similar to instance method definitions; the only difference is that in place of both the return type and method name (as in `String getAuthor`) is simply a repetition of the class name.

With the two constructors defined as in Figure 9.1, we can now create Book objects.

```
theOdyssey = new Book("Homer", "The Odyssey");
myBook = new Book(theOdyssey);
```

When defining a class, good programmers will tag each element of the class with **`public`** or **`private`**, indicating whether this is something that other classes should be able to access. Nearly all instance variables are **`private`** and most instance methods are **`public`**. There may, though, be some methods that are defined only to help with other methods; since there is no reason to allow other classes to use such methods, a good programmer would tag it as **`private`** so that it could later be changed without worrying about whether other classes use it.

In fact, we will *always* tag instance variables as **`private`** in this course, and you should do this also in your coursework. If you fail to tag an instance variable as **`private`**, the program may still run correctly, but it could be graded as if you made a mistake.

Figure 9.2 illustrates a complete definition of the Book class.

Figure 9.2: Defining the `Book` class: Take three.

```
public class Book {
    private String author;
```

```
    private String title;

    public Book(String inAuthor, String inTitle) {
        author = inAuthor;
        title = inTitle;
    }

    public Book(Book original) { // creates copy of original book
        author = original.author;
        title = original.title;
    }

    public String getAuthor() { return author; }
    public String getTitle()  { return title;  }

    public boolean isCopy(Book query) {
        return author.equals(query.author) &amp;& title.equals(query.title);
    }
}
```

Sometimes, in writing a method, it's useful to have variables that temporarily remember information for the duration of the method. For example, we might instead want to write our `isCopy` method thus.

```
public class Book {
    // other stuff...
    public boolean isCopy(Book query) {
        boolean authorMatch;
        boolean titleMatch;
        authorMatch = author.equals(query.author);
        titleMatch = title.equals(query.title);
        return authorMatch && titleMatch;
    }
}
```

The `authorMatch` and `titleMatch` variables are *local variables*: They exist only locally within the `isCopy` method, not available to other methods. Note that the declaration looks identical to instance variable declarations; the only difference is in its location: A local variable is declared within a method, whereas an instance variable declaration occurs outside methods' definitions. Local variables are different from instance variables in that

local variables will vanish as soon as the method is complete. I like to think of local variables as an object's short-term memory — the difference between jotting something down on scrap paper and writing something into a planner. You should use local variables whenever possible.

Classes can also contain *class variables* and *class methods*. Whereas every instance has its own version of the instance variable, a class variable is associated with the overall class (of which there is only one). Similarly, a class method is something you tell the overall class to do. Class variables and class methods are defined just like instance variables and instance methods except that their definitions include the keyword ***`static`***.

To summarize, Java has three categories of variables and three categories of methods.

variables	***methods***
instance variable	instance method
local variable	constructor
`static` variable	**`static`** method

Local variables must be defined within a method; all others must appear directly within a class definition. While Java doesn't strictly insist on any order for what is defined within the class, people generally adopt some rule and stick with it. In this book, all class definitions begin with instance variables, then constructors, and finally instance methods.

9.2. Inheritance

Sometimes, it happens that there are some classes that are really more sophisticated versions of other classes. For example, our library may contain recordings of books, and perhaps the card catalog should contain the playing time of each recording. Since a recording is just a specialized book, we will define the Recording class as one that *extends* Book.

```
public class Recording extends Book {
    private int playingTime;

    public Recording(String inAuthor, String inTitle, int inPlayingTime) {
        super(inAuthor, inTitle);
        playingTime = inPlayingTime;
    }
```

```
    public int getPlayingTime() { return playingTime; }
}
```

While the definition doesn't say it explicitly, every Recording has all the Book instance variables — `author` and `title`; these properties are said to be *inherited* from the Book class. (Of course, as defined above, each Recording has a `playingTime` instance variable also.) The instance methods are inherited, too, but not the constructors: The only constructor that can be used to create a Recording are those defined in the Recording class.

In such a program, Recording is called a *subclass* of the Book class, while Book is a *superclass* (or *parent class*) of Recording. People would say that the Recording class*extends* the Book class.

Java regards any Recording object as a Book object also. Thus, we can assign a Book variable to refer to a Recording instance, as in the following.

```
Book myBook = new Recording("Homer", "The Odyssey", 959);
```

This is only possible because Recording extends the Book class. The statement would not be legal otherwise, even if Recording happened to have all the instance variables and instance methods of the same names.

9.3. Abstract classes and interfaces

Sometimes it makes sense to have a class whose only instances are instances of a subclass. Our library, for example, may have a variety of items, including films, artwork, and magazines, which are not books. But we might want an array of all such items in the library, and all of the items may have some variable or method in common, such as a title attribute. To support this, we can define a class called *Holding*, which would have Book, Film, and ArtPiece all as subclasses. While we might want a Holding variable to refer to any of these, there is no reason that we would want to create a raw Holding object. Such a class is an *abstract class*, and we define it using the word **`abstract`** in its definition.

```
public abstract class Holding {
    private String title;

    public Holding(String inTitle) {
        title = inTitle;
    }
```

```
    public String getTitle() { return title; }
}
```

Defining the Holding class as ***abstract*** prevents the user from constructing Holding objects; the expression **new** `Holding("Iliad")` is illegal. But we can define subclasses.

```
public class Book extends Holding {
    private String author;

    public Book(String inAuthor, String inTitle) {
        super(inTitle);
        author = inAuthor;
    }

    public String getAuthor() { return author; }
}
```

Since the Book subclass isn't abstract, we can create Book instances. Of course, since Book is a subclass of Holding, any Book object can act as Holding instance: That is, a Holding variable can legally refer to what is in fact a Book object.

It often happens with abstract classes that all instances of the abstract class must have a method, but the method cannot be defined in the abstract class because not enough information is there. For example, we might want every holding to be able to tell its proper location within the library, but the location will depend on the nature of the item. For this, the *abstract method* is useful. The following illustrates the definition of an abstract method.

```
public abstract class Holding {
    // ... other elements ...
    public abstract String getLocation();
}
```

Note that we use the word ***abstract*** again in the method definition, and we use a semicolon in place of defining the method's body. With this definition, every non-abstract subclass must define the method in order to be compiled. We can call `getLocation` on a Holding variable, and it will use the method that is defined for the instance to which the variable happens to refer: Book's `getLocation`, if the object happens to be a Book, or if it is a film, Film's `getLocation` method.

The concept of *interface* is related: An *interface* is a collection of several instance methods, which must all be abstract. It cannot contain constructors, class methods, or variables.

(Constants — i.e., class methods that are declared as `final` — are an exception.) For example, we might want an Authored interface for items that have an author.

```
public interface Authored {
    public String getTitle();
    public String getAuthor();
}
```

Individual classes, like Book, can be declared to implement an interface, as long as they also define all the methods in the interface.

```
public class Book extends Holding implements Authored {
    // ...
```

Variables of the interface type are allowed. We might have an Authored variable, for example, and it would be allowed to refer to a Book object or any instance of another class implementing the Authored interface.

Other classes that don't define all of the methods can't be declared as implementing the Authored interface. The Film class, for example, probably doesn't have a `getAuthor` method, so it couldn't be said to implement the Authored interface. Notice that the Book class both extends the Holding class and implements the Authored interface. Classes can implement any number of interfaces, but each can extend only one class.

The distinction between interfaces and abstract classes sets up a tradeoff between the two: Abstract classes can include instance variables and full method definitions, whereas interfaces are allowed to include only abstract instance methods. However, a class can implement any number of interfaces, but it can only extend one abstract class. In practice, while the two concepts are similar, there isn't too much confusion about whether to use one or the other.

Source: http://www.toves.org/books/data/ch09-rev/index.html