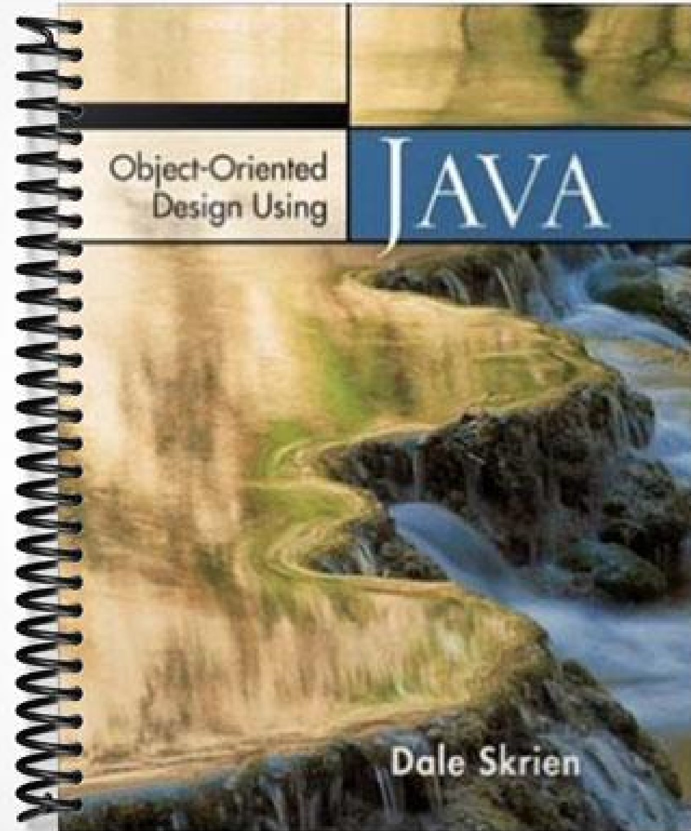
SOLUTIONS MANUAL
Object-Oriented
Design Using
JAVA
Dale Skrien

Chapter 2

Exercises and Solutions

1. Briefly describe how OO programming, as compared to non-OO programming, makes it easier to write elegant software, as discussed in Section 1.3 of Chapter 1.

   usability – OO programs are as usable as other kinds of code—it doesn't really make a significant difference here

   completeness – ditto

   robustness – OO doesn't affect robustness significantly, unless the particular OO language has features designed to help you handle unusual situations by, for example, throwing exceptions

   efficiency – OO languages are sometimes a little slower than non-OO languages (from method call overhead) but not enough to make a huge difference in most programs.

   scalability – OO code can be as scalable or not scalable as other code. Scalability is often affected by the algorithms and data structures you chose, regardless of the language you use.

   readability – OO is often a more natural way to think about a problem, which makes code much more readable. Don't be mislead, though; it's still very easy to write an unreadable program in an OO language.

   reusability – OO is good for this; you easily separate pieces of the program, namely classes, and apply them to different tasks in other programs.

   simplicity – It depends on what you're writing. Large OO programs can be as simple or complex as large non-OO programs, but for small programs, it can seem unnecessarily complicated to go through the bother of creating a class (say for a simple "Hello World" program) in languages such as Java. Other OO languages are better at keeping small programs simple.

   maintainability – OO is pretty good at this; it comes with modularity of OO code; if things are crafted into separate classes and packages, it is far easier locate and fix/replace the part that isn't working. If code is not modular, then an error can propagate itself into places you would never expect, and fixing it becomes an annoyingly long chore of tracking down all the code that the error affected.

   extensibility- OO languages are great at extensibility. Inheritance naturally gives extensibility.

2. In Section 2.2, a Person class is defined and then a Person object is created by the following line of code:

   ```
   Person firstPerson = new Person("Adam", new Date(0));
   ```

   In what month and year and on what day is Adam's birth date? Hint: Look at the documentation of the Date class in the java.util package.

   Adam was born on Jan 1, 1970, 00:00:00 GMT (assuming of course, that the Date in the constructor is his birthday).

3. Give a specific example, other than the example of constants, where it would make sense to have a class variable (that is, a variable declared "static"). That is, give an example where it would make sense for all objects of a class to share a variable.

   Suppose you had a Person class, and you wanted to calculate the population based on how many Person objects had been instantiated. One way of keeping track of the total population is

to have a class variable called population, and to have the constructors update that variable every time it is called.

4. In Section 2.3, we mentioned that the Set class' static method `intersect(Set s1, Set s2)` can treat a null `s1` or `s2` as an empty set and so return an empty set. Assuming we really want a static method for determining set intersections (rather than an instance method), is this particular behavior with null parameters a good design? Why or why not?
This is not a good design because it is unlikely that you will ever want to represent an empty set as a null pointer. If the method sees a null pointer, it is much more likely that it represents a variable that hasn't been initialized yet (i.e., a bug) than an empty set. Therefore, the considerate thing to do would be to throw an exception and let the programmer know that they (probably) have a bug, rather than pretending that nothing is wrong and giving them back some answer that is probably useless.

5. Look in the Java APIs for the javax.swing.JOptionPane class. It contains many class methods. Why would the Java class designers choose to make them class methods instead of instance methods?
If these methods were instance methods, you would have to create a JoptionPane object. However, the JOptionPane object plays no role, in that its fields are never used. Instead, the methods are self-sufficient and so it is appropriate to make them static.

6. In the Rectangle example in Section 2.5, the programmer desired a class like Rectangle except with `getCenter` and `setCenter` methods, and so the programmer subclassed Rectangle. An alternative solution to the programmer's problem is to use the Rectangle class and wherever a call to `setCenter` or `getCenter` is desired, instead "inline" the call. That is, instead of calls to `getCenter` and `setCenter`, there could be code to get and set the center manually. For example, instead of
```
Point center = r.getCenter();
```
the programmer could write
```
Point center;
center.x = r.getCenterX();
center.y = r.getCenterY();
```
which accomplishes the same objective. Then the programmer wouldn't have to subclass Rectangle. Discuss whether this solution is preferable to the solution involving subclassing.
The solution involving subclassing is slightly more elegant because it allows the Rectangle more control in how the center is calculated in that the actual code that gets or sets the center is encapsulated into the body of the `getCenter` and `setCenter` methods. But the main factor is readability: Writing "`getCenter`" and "`setCenter`" is clearer to the reader. Another factor in choosing between them is how much code you have to write. Why duplicate those lines of code over and over when you can just call `getCenter` or `setCenter` instead?

7. In Section 2.5 we mentioned that you can use implementation inheritance to create a subclass that specializes a superclass either by adding some new behavior or by changing existing behavior. We can also use implementation inheritance to *remove* some existing behavior. Explain how it would be done and discuss the elegance or inelegance of this use of inheritance.

This can be done simply by overriding a superclass method and, if it is a void method, giving it an empty body, or, if it is a function, just returning a default value. Another alternative is to throw an Exception, such as a MethodNotImplementedException. However, this is not very elegant; no one expects a method to do nothing, so the user will be rightfully surprised when their program behaves strangely. In general principles, an overridden method should do everything that the other one does, and possibly more.

8. In Section 2.5, we introduced HumanCustomer, BusinessCustomer, and Customer classes, but no complete implementations of such classes were given. Implement all 3 classes. Assume that, in addition to a name, the HumanCustomer class has a spouse and a list of children as attributes with `getSpouse` and `GetChildren` methods and that a BusinessCustomer has a size attribute giving the number of employees and a `getSize` method. Don't forget to create appropriate constructors for all three classes.

9. In the Customer example of Section 2.5, the Customer class was created to avoid code duplication in the BusinessCustomer and HumanCustomer classes. An alternative approach that also avoids such duplication is to combine the two classes into one Customer class that includes all the instance variables and methods of both classes. There would also be an extra boolean variable `isHuman` that is true if the Customer corresponds to a human and is false if the customer is a business. Using the value of this extra variable, you could decide how to respond to any request (i.e., any method call) regarding the customer. What do you think of this implementation? Explain.
   Using the `isHuman` method is roughly as elegant as using the two-class scenario with a lot of `instanceof` calls, because you're determining the type of object in a nonextensible way. If later a third kind of customer is created, it will be difficult to extend the functionality of this class in a concise and elegant manner to handle the new customer. Basically, you will probably have to add an `isX` method for every new kind of customer, and you'll have a lot of large if-statement blocks looking like `if (foo.isX()){...}else if(foo.isY()){...}else if...` If you can guarantee that no further kinds will ever exist (which is really hard to do), then I'd say it's an acceptable approach (but still not a great one), but otherwise subclassing is preferable.

10. In Section 2.5, we talked about using inheritance to create a common superclass to two subclasses and then move duplicated code into the superclass. Following this reasoning, one might be tempted to create a common superclass of classes Pet, Person, Street, City, and Country objects, if they all have a `getName` method, to avoid duplication of code. If there were such classes, would a superclass be appropriate here? Briefly explain. If it is appropriate to have such a superclass, what would you call the superclass and should it be abstract? More generally, when is it inappropriate to create a common superclass of classes with common behavior or attributes?
    Putting common code into one place to avoid duplication is always better than having duplicated code, however in this case there is very little gained by doing so. Creating a superclass, (let's call it "Nameable") is better than not having a superclass, but creating a superclass like "Place" for Street, City, and Country would be even better than having Nameable for all of them, because you would probably be able to put other common code into the Place class. Another problem is that classes can have at most one superclass. The two

classes might need different superclasses for other reasons. If you do create a Nameable superclass, it should be abstract because you don't want to create objects of that class, but rather objects of its subclasses.

11. In Section 2.5, we introduced a FilledOval class whose `draw` method overrode the parent's `draw` method. However, the Oval's instance variables are public, which, as we later discussed, is not optimal. Change those instance variables to private and reimplement any code that depended on the public accessibility of those instance variables.

```
import java.awt.*;
public class Oval
{
   private int x, y, w, h;

   public Oval(int x, int y, int w, int h) {
      this.x = x; this.y = y; this.w = w; this.h = h;
   }

   public Oval(Point topLeft, int w, int h) {
      this.x = topLeft.x;
      this.y = topLeft.y;
      this.w = w;
      this.h = h;
   }

   public void draw(Graphics g) {
      g.drawOval(x, y, w, h);
   }

   public int getWidth() { return w; }

   public int getHeight() { return h; }

   public Point getTopLeftPoint() { return new Point(x,y); }

   public void setWidth(int w) {
      this.w = w;
   }

   public void setHeight(int h) {
      this.h = h;
   }

   public void setTopLeftPoint(Point p){
      this.x = p.x;
      this.y = p.y;
   }

   //...other methods...
}

public class FilledOval extends Oval
```

```
{
   public FilledOval(int x, int y, int w, int h)
   { super(x, y, w, h); }

   public void draw(Graphics g)
   {
      Point tl = this.getTopLeftPoint();
      g.fillOval(tl.x, tl.y, this.getWidth(), this.getHeight());
   }
}
```

12. In Section 2.6, we mentioned that, to replace a LinkedList object in the package with an ArrayList object, it is not sufficient to do a global search and replace. Give one reason why this approach is insufficient.
    Here are several reasons. For starters, the interfaces of the two classes are not exactly the same, so you'll have to change some method calls to get it to work right. For example, the LinkedList class has an `addFirst` method and the ArrayList class has no such method. Therefore, any use of `addFirst` will need to be replaced with something equivalent in the ArrayList class. Also, you may want LinkedList objects some places and not other places, in which case a global search and replace is not appropriate. Finally, there may be other LinkedList classes in other packages in your system, and you wouldn't want them replaced.

13. In Section 2.6, we said that "if software is elegant, then one should be able to remove an object of one class and easily replace it with, or 'plug in', an object of another 'equivalent' class either on the fly or with only minimal code changes". Explain what "equivalent" and "minimal" mean based on the discussion in Section 2.6.
    In this case "equivalent" means "with the same interface" and "minimal" means "one". Elegant code will, wherever possible, refer to interfaces instead of actual classes and use factories to create the objects implementing an interface. In such code, to replace an object of one class with an object of another class that implements the same interface, you only need to make one change, namely change the kind of object returned by the factory's creation method.

14. In Section 2.6, we talked about the problems of changing type of list in our application. In our discussion, we assumed that once the type of list was chosen, it remained fix for the duration of the execution of the application. However, what if we wanted our application to be able to change type of list on the fly? That is, during execution, the user or the program itself might decide that we need to start using ArrayLists instead of LinkedLists for all lists that will be created in the future (we will allow existing lists to remain as is). Tell how this would be implemented (a) using a factory method and (b) without using a factory method. Is the factory method version more elegant? Explain.
    With a factory method, you would have some class like "ListFactory" with a method like `public List getList()`. Each time you called `getList`, it would just give you the right kind of list and you would use it and go on. To change it on the fly, the user would need some sort of method the allowed him to set which type of List would be returned. Perhaps the easiest way to do so is to add a parameter to the ListFactory's `getList` method. For example, the method could be `List getList(String listType)` where `listType` is the name of

a class implementing the List interface. The `getList` method could use the java.lang.reflect classes to create a list of type `listType` and then return it.

Not using a factory is much less elegant. The idea behind a factory is that you have one place (typically a class or object) that controls the decision-making process for the construction of a certain kind of Object. Not using a factory implies that each bit of code that wants to make a list must figure out for itself what sort of list to make. Basically this implies code duplication, especially if you're going to be making lots and lots of Lists all over the place. The decision-making process should be essentially the same (you're still going to need some way of indicating what kind of list to make), it would just be repeated time and time again throughout the code.

15. In Section 2.8, we gave the Automobile class' `getCapacity` method a default behavior of returning 0 to help us explain how dynamic method invocation and overriding works. However, this design is not very elegant, and, in fact, the Automobile should be an interface. Briefly explain why.

    The design is inelegant because it gives a default behavior where none should be given. If someone subclasses Automobile, they could potentially forget to override getCapacity, and their new class would give the confusing result that it could not hold any people.

16. Consider an Automobile class with the following implementations of `equals` methods:

```
public boolean equals(Object o) {…}  //version A, inherited from Object
public boolean equals(Automobile o) {…}  //version B
```

and a class Sedan that is a subclass of Automobile with the following methods:

```
public boolean equals(Automobile o) {…}  //version C
public boolean equals(Sedan o) {…}  //version D
```

and a class Minivan that is a subclass of Automobile with the following methods:

```
public boolean equals(Automobile o) {…}  //version E
public boolean equals(Minivan o) {…}  //version F
```

Now suppose you have the following 6 variables:

```
Object o = new Automobile();
Automobile auto = new Automobile();
Automobile sedanAuto = new Sedan(Color.black);
Automobile minivanAuto = new Minivan(Color.blue);
Sedan sedan = new Sedan(Color.grey);
Minivan minivan = new Minivan(Color.pink);
```

There are 36 ways each of these variables can be paired off in a call to `equals`, where one of the 6 variables is being sent the `equals` message and one of the 6 is the argument to `equals`. For each such pair, indicate which of the `equals` methods will be executed. For example, if there is a call

```
auto.equals(auto)
```

then version B of `equals` will be called. Figure out which `equals` method will be executed for the other 35 combinations.

```
o.equals(o)                 A
o.equals(auto)              A
o.equals(sedanAuto)         A
o.equals(minivanAuto)       A
o.equals(sedan)             A
o.equals(minivan)           A
auto.equals(o)              A
```

```
auto.equals(auto)                  B
auto.equals(sedanAuto)             B
auto.equals(minivanAuto)           B
auto.equals(sedan)                 B
auto.equals(minivan)               B
sedanAuto.equals(o)                A
sedanAuto.equals(auto)             C
sedanAuto.equals(sedanAuto)        C
sedanAuto.equals(minivanAuto)      C
sedanAuto.equals(sedan)            C
sedanAuto.equals(minivan)          C
minivanAuto.equals(o)              A
minivanAuto.equals(auto)           E
minivanAuto.equals(sedanAuto)      E
minivanAuto.equals(minivanAuto)    E
minivanAuto.equals(sedan)          E
minivanAuto.equals(minivan)        E
sedan.equals(o)                    A
sedan.equals(auto)                 C
sedan.equals(sedanAuto)            C
sedan.equals(minivanAuto)          C
sedan.equals(sedan)                D
sedan.equals(minivan)              C
minivan.equals(o)                  A
minivan.equals(auto)               E
minivan.equals(sedanAuto)          E
minivan.equals(minivanAuto)        E
minivan.equals(sedan)              E
minivan.equals(minivan)            F
```

17. In Section 2.9, we mentioned two `substring` methods in the String class. Suppose it was your job to implement these two methods. Describe how you would go about avoiding code duplication in the bodies of these two methods.

The `substring(int beginIndex)` can simply be a call to `substring(int begin, int end)`. It would look something like this

```
public String substring(int begin){
   return this.substring(begin, this.length());
}
```

18. In Section 2.10, we gave the example of a SSNWrapper class that allowed subclasses to modify the `socialSecurityNumber` instance variable through a protected modifier method. Wouldn't it be a whole lot easier to just make `socialSecurityNumber` itself protected and so avoid the need for the protected `setSSN` method in the SSNWrapper class? In that case, the code would look like this:

```
public class SSNWrapper
{
   protected int socialSecurityNumber;

   public SSNWrapper(int ssn) { socialSecurityNumber = ssn; }
```

```
      public int getSSN() { return socialSecurityNumber; }
   }

   public class SettableSSNWrapper extends SSNWrapper
   {
      public SettableSSNWrapper(int ssn) { super(ssn); }

      public void setSSN(int ssn) { socialSecurityNumber = ssn; }
   }
```

By making this change, we've eliminated the need for the protected method in the SSNWrapper class. Is this version better or worse than the version in the text? Explain.
This is less elegant in much the same way as making a variable public is. Whether you use a public/protected method or a public/protected field, you're still giving the same functionality to the same classes. The difference is that you can override a method in a subclass, but not a field. Also, if you use the method, you can still keep some control of how the fields are used within the class. For example, what if you later want to change the implementation of SSNWrapper so that it stored the SSN in a string or some other form such as 3 integers? You can do it with the first approach and subclasses won't be affected, but you can't in the second approach.

19. In the java.awt.event package, there is an ActionListener interface that many classes implement when they want to be made aware of events that occur, such as mouse clicks in buttons. Why is ActionListener an interface instead of an abstract class? How would making ActionListener an abstract class decrease the usability of the class?
If ActionListener were an abstract class, then it would “use up” the superclass of those classes wanting to be ActionListeners. More importantly, there is no reason for the ActionListener to provide any implementation, and so an interface is better.

20. For each of the following pairs of classes, tell which class of the pair should be a subclass of which, if either. Briefly explain why or why not.
(a) Car and Tire
(b) Car and Truck
(c) Card (with suit and value instance variables) and Deck (of 52 cards)
a – neither. Car should have a tire (or 4), but a tire is not a car nor is a car a tire.
b – It depends on what you mean by a car. If Car means Automobile (some wheels, an engine, and so forth), then yes, Truck would be a good subclass of Car, because a Truck is a Automobile/Car with some added functionality (like a truck bed) and so inheriting all of the Car’s features makes sense. However, if Car means sedan, then Truck would not be a good subclass, because a sedan-Car is much more specific (4 doors, no bed, a trunk, back seats, etc.), and so inheritance makes no sense. In this case, a Car would not make a good subclass of Truck, either.
c – neither. A Deck is a collection of 52 cards, but a Deck is not a Card, and neither is a Card a Deck. Inheritance makes no sense.