Quiz yourself: Apply encapsulation principles to a class

Test your knowledge of a key object-oriented programming concept.

by Simon Roberts and Mikalai Zaikin

October 27, 2020

If you have worked on our quiz questions in the past, you know none of them is easy. They model the difficult questions from certification examinations. We write questions for the certification exams, and we intend that the same rules apply: Take words at their face value and trust that the questions are not intended to deceive you but to straightforwardly test your knowledge of the ins and outs of the language.

Given this `BookOrder` class:

```java
public class BookOrder {
    private int id;
    private List<String> items = new ArrayList<>;
    public BookOrder(int id) {
        this.id = id;
    }
    public int getId() {
        return id;
    }
    public List<String> getItems() {
        return items;
    }
    public String toString() {
        return String.valueOf(id) + " " + items;
    }
}
```

Which statement correctly characterizes class encapsulation? Choose one.
A. The class is properly encapsulated because its instance variables areprivate.
   The answer is A.

B. The class is not properly encapsulated because client code can arbitrarily change object state.
   The answer is B.

C. The class is not properly encapsulated because the implementation of theidvariable allows the creation of orders with the same ID.
   The answer is C.

D. The class is not properly encapsulated because encapsulation implies getters and setters, and there are no setter methods.
   The answer is D.

Answer. Encapsulation is one of several object-oriented programming concepts called out specifically in the exam objectives.

A major goal of encapsulation is to allow objects to protect themselves against arbitrary manipulation of their state, particularly manipulation that might result in invalid state. Invalid state is easily exemplified by the idea of a date that has values that make no sense, such as month -3 or day of month 42.

Encapsulation in Java is generally implemented by making instance variablesprivateand providing indirect access using methods (sometimes simply getters and setters) to read and manipulate them. Those methods that mutate the internal state should implement checks to ensure that the result of the operation will still be a valid internal state.

At first glance, option A looks promising because any mutable data member probably needs to beprivate to have a chance at being protected from being modified to an invalid value. However, while making fields private is necessary, it's not always sufficient to prevent unwanted effects. In the code in the question, there is no direct mechanism provided to mutate theList—there is no “setter” method as such—but theListcan be easily manipulated and even completely cleared without your control by malicious client code. This can be done, for example, simply by calling the methodgetItems, and then directly manipulating theListthat is returned. For example, consider this incomplete code:

```java
BookOrder bo = ... // obtain an arbitrary order
List<String> books = bo.getItems(); // gets r
books.clear(); // removes all current entries
books.add("Java For Dummies"); // let's add n
books.add("Java For Smarties"); // add a very`
After running this code, the original books will have been removed from the order, and two new ones will have been added to it.

From this, you can conclude option B is correct and option A is incorrect.

Several possible solutions exist for this problem. One is to remove the method that returns the list and instead add methods to allow the caller to request changes to the internal list. (Presumably these would verify whether to permit the change and also enforce any necessary consequences, such as updating the total price of the order.)

Another is simply to return an immutable proxy that allows reading the list items but prevents changes, though this supposes that other features are added to allow initialization, and perhaps mutation, of the contents under proper control of the order. A feature of this kind is built into the core Java APIs and might be implemented using the following modified `getItems` method.

```java
public List<String> getItems() {
    return Collections.unmodifiableList(items);
}
```

Option C is incorrect. It is true that the current class implementation permits creating several orders with the same ID, but whether this should be allowed is a problem-domain question and not specifically a matter of encapsulation. Indeed, if, as is likely, these IDs should be unique to a particular order, that doesn’t mean you will never want to create two objects representing the same order. These might be required at different points in time or in different subsystems. But to the point of the question, the `id` variable is well encapsulated; it cannot be altered after the object is initialized.

Option D is also incorrect. Setters can be part of a well-encapsulated object, but encapsulation is about preventing unconstrained changes to fields. Having no way whatever to change a field is surely the most rigorous protection against unconstrained changes and, therefore, it does not break encapsulation. In fact, there is a growing movement towards the idea, taken from functional programming style, that immutable objects are probably better for creating comprehensible and thread-safe code.

**Conclusion:** The correct answer is option B.

---

**Simon Roberts**

Simon Roberts joined Sun Microsystems in time to teach Sun’s first Java classes in the UK. He created the Sun Certified Java...
Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who publishes recorded and live video training through Pearson InformIT (available direct and through the O’Reilly Safari Books Online service). He remains involved with Oracle’s Java certification projects.

Mikalai Zaikin

Mikalai Zaikin is a lead Java developer at IBA IT Park in Minsk, Belarus. During his career, he has helped Oracle with development of Java certification exams, and he has been a technical reviewer of several Java certification books, including three editions of the famous Sun Certified Programmer for Java study guides by Kathy Sierra and Bert Bates.