Lecture 10: Access Modifiers and Encapsulation

10:00 AM, Feb 14, 2018

Contents

1 Code Critique

2 Encapsulating Knowledge
   2.1 Putting Methods in Their Proper Place
   2.2 Access Modifiers in Java
      2.2.1 Guidelines on Access Modifiers
   2.3 Adapting the Banking Service to Access Modifiers

3 Communicating History Between Methods

4 Encapsulating Representation
   4.1 Replace Fixed Data Structures with Interfaces
   4.2 Create Concrete Classes that Implement the New Interfaces
   4.3 Initialize Data with Objects of the New Concrete Class
   4.4 Anticipated Questions

5 Summary
   5.1 Two Myths About Encapsulation
   5.2 The Many Faces of Variables

Objectives

By the end of this lecture, you will know:

- about access modifiers in Java
- how access modifiers impact where computations can be performed in Java
- about uses of variables to coordinate computation across methods

By the end of this lecture, you will be able to:

- use Java’s access modifiers to control access to fields
modify code to move methods into classes as appropriate

So far, we’ve focused on how to create classes that are amenable to future extensions. Today, we look at making code robust, both in how to enable future modifications (as well as extensions) and in protecting against malicious or unintentional programming errors.

1 Code Critique

For this lecture, start with the following starter file for a banking service. Critique it: what problems do you see in this code with regards to future modifications or information protection?

http://cs.brown.edu/courses/csci0180/content/sources/BankingInit.java

1. Any class that has access to a customer object has the ability to access or change that customer’s password. In the BankingService class, for example, the login method directly accesses the password to check whether it is valid; that method could just as easily (maliciously!) change the password. The contents of the password should never get out of the Customer class.

   The real problem here is that login should be a method on Customer, which has the data that the method needs.

2. A similar concern applies to the balance field in withdraw, but withdraw illustrates another problem. Imagine that the bank adds more details to accounts (such as overdraft protection or a withdrawal fee). The BankingService class would have to keep changing as the notion of Accounts changes, which makes no sense. The BankingService class simply wants a way to execute a withdrawal without concern for the detailed structure of Account objects. The withdraw method needs to be a method on Account, not BankingService.

3. The BankingService class has written all of its code over accounts and customers against a fixed data structure (the LinkedList). The dependency is clear in the code for the methods (getBalance, withdraw, and login): each includes a for-loop over the list in its implementation. What if we wanted to replace the LinkedList with something more efficient to traverse, like an array?

4. The dummy return value of 0 in getBalance and withdraw is awful, because it does not distinguish between a valid answer (an account balance of 0) and an error condition. Picking a dummy value to satisfy the type system is never a good idea. This program needs a better way of handing errors. We will get back to this in a couple of weeks.

5. The withdraw method doesn’t require the account holder to be logged in. There’s no point having a login capabilities and not actually using them.

Underlying the first three of these concerns is a goal called encapsulation. Intuitively, encapsulation is about bundling data and code together in order to (1) reduce dependencies of one part of a system on structural details of another, and (2) control manipulation of and access to data. This lecture is about recognizing where encapsulation is needed and learning how to introduce it into your program. The next lecture will address error handling (item 4).
2 Encapsulating Knowledge

Problems 1 and 2 are fundamentally failures to keep data and methods on them in the same class. Here, encapsulation is about regulating access to data (for purposes of reading, modifying, or even knowing about the existence of some data). These problems illustrate why we want to encapsulate knowledge in programs.

We will fix these problems in two stages: first, we move each method into its proper class (and rewrite the BankingService to use the new methods; second, we protect the data within these classes from unauthorized access.

2.1 Putting Methods in Their Proper Place

Let’s move the withdraw and getBalance methods into the Account class:

```java
public class Account {
    int number;
    Customer owner;
    double balance;

    // returns the balance in this account
    double getBalance() {
        return this.balance;
    }

    // deducts given amount from account and returns total deduction
    // if add account info, no need to edit BankingService
    double withdraw(double amt) {
        this.balance = this.balance - amt;
        return amt;
    }
}
```

Methods like getBalance, which simply return the value of fields, are called getters. Many OO books suggest adding getters (and a corresponding setter method to change the value) on all fields. This guideline is too extreme though—we’ll return to it at the end of the lecture.

The getBalance and withdraw methods in the BankingService class change as follows to use the new methods. Note that neither one now directly accesses the field containing the data in Account.

```java
double getBalance(int forAcctNum) {
    for (Account acct:accounts) {
        if (acct.number == forAcctNum)
            return acct.getBalance();
    }
    return 0;
}

double withdraw(int forAcctNum, double amt) {
    for (Account acct:accounts) {
        if (acct.number == forAcctNum) {
            return acct.withdraw(amt);
        }
    }
}
```
One advantage to having the separate withdraw method in the Account class is that if the data in an account changes, we can change the withdrawal computation without affecting other classes. For example, if the bank introduced withdrawal fees, then the amount deducted from an account would be the amount requested plus the fee. This new code structure lets the BankingService simply ask to perform the withdrawal, leaving the specifics to the Account class.

Next, let’s move login into the Customer class. The result is similar.

```java
public class Customer {
    String name;
    String password;
    LinkedList<Account> accounts;

    // check whether the given password matches the one for this user
    // in a real system, this method would return some object with
    // info about the customer, not just a string
    String tryLogin(int withPwd) {
        if (this.password.equals(withPwd))
            return "Welcome";
        else
            return "Try Again";
    }
}

public class BankingService {
    ...
    String login(String custname, int withPwd) {
        for (Customer cust:customers) {
            if (cust.name.equals(custname)) {
                cust.tryLogin(withPwd);
            }
        }
        return "Oops -- don't know this customer";
    }
}
```

### 2.2 Access Modifiers in Java

Even though we have edited the BankingService to not directly access a customer’s password or the balance in an account, nothing we have done prevents the BankingService (or a future extension of it) from doing so. To make this program more robust, we want to protect the data in the Customer and Account classes from direct access or modification from outside classes. Other classes may be able to access or modify these through getters, setters, or other methods, but at least then the programmer providing those methods has some control over how the access occurs. The question, then, is how to prevent direct access to the fields of a class using an ”object.field” expression.
Java provides several *access modifiers* that programmers can put on classes and methods to control which other classes may use them. The modifiers we will consider in this course are:

- **private** means the item is only accessible by name inside the class. If you make a field private, for example, then if you wanted an object from another class to access the field, you would need to provide a method (like a getter) that enables the access.

- **public** means every other class or object can access this item.

- **protected** means that objects in the current class and all of its subclasses (and their subclasses) can access this item.

If you don’t put any annotation on a field or method, it is *package-private*, which means it is accessible anywhere within the package, but not outside the package.

There are some additional ones that you can use when organizing Java code into larger units called packages; you’ll get to those if you take Software Engineering.

For our banking application, we want to make all of the fields in all of the classes private. This is a good general rule of thumb, unless you have a good reason to do otherwise. In addition, you should mark methods meant to be used by other classes as *public*. Concretely, the `Customer` and `Account` classes now look like:

```java
class Customer {
    private String name;
    private int password;
    private LinkedList<Account> accounts;

    public String tryLogin(int withPwd) {
        ...
    }
}

class Account {
    private int number;
    private Customer owner;
    private double balance;

    public double getBalance() {
        ...
    }

    public double withdraw(double amt) {
        ...
    }
}
```

Access modifiers are checked at compile time. Try accessing `cust.password` in the `login` method in `BankingService` with the access modifiers in place to see the error message that you would get.

Now that we’ve seen access modifiers, we can explain why Java requires that methods that implement interfaces are marked *public*. The whole idea of an interface is that it is a guaranteed collection of
methods on an object. The concept would be meaningless if the methods required by an interface were not public. The fact that you get a compiler error without the public, though, suggests that public is not the default modifier. That is correct. The default modifier is "public within the package" (where package is the concept you will see in SoftEng for bundling classes into larger units). This is more restrictive than pure public, so the public annotation is required on all methods that implement parts of interfaces.

2.2.1 Guidelines on Access Modifiers

Good programming practice recommends the following guidelines:

- Put access modifiers on every field and method (including constructors) in a class.
- Make all fields private unless another guideline applies.
- Any method that is visible through an interface must be public.
- Any method that another class must use should be public.
- Any field/method whose visibility can be limited to subclasses should be marked protected.
- Make constructors in abstract classes protected, so subclasses can invoke them.
- Make constructors that can’t be used by other classes private.

Note that subclasses cannot make something less visible than in their superclass. So if a class declares a field as public, you cannot extend the class and have the field be private in the extended class. The reasons for this have to do with the inconsistency of access information given that Java can view an object as either a member of its class or any superclass of its class.

2.3 Adapting the Banking Service to Access Modifiers

Now that we’ve made the Customer and Account fields private, our code doesn’t compile. We had references to cust.name in the login method, for example; those are not allowed on private fields. To fix the code, we need to put a method in the customer class (which can access the name). Two options come to mind:

- Put a getName method in Customer to return the name, then replace cust.name with cust.getName().
- Put a nameMatches method in Customer that takes the name to compare to and checks whether the names are equal.

The first suggestion is called a "getter" in object-oriented programming. Many tutorials will suggest making a getter for all fields. Is this a good idea?

No. Ask yourself why you made the name private in the first place. If you wanted to keep people from reading it, a getter circumvents that decision. If you only wanted to keep people from modifying
it, a getter might make sense, but good OO practice will still recommend the second approach – a method that performs the computation that you actually want to see happen on the data.

If we follow the second approach, our `Customer` class will look at follows:

```java
public class Customer {
    private String name;
    private int password;
    private LinkedList<Account> accounts;

    // check whether customer has given name
    public boolean nameMatches(String aname) {
        return (this.name.equals(aname));
    }

    // produce message based on whether given password matches
    public String tryLogin(String withPwd) {
        if (this.password.equals(withPwd))
            return "Welcome";
        else
            return "Try Again";
    }
}
```

with the `login` method in `BankingService` updated to:

```java
public String login(String custname, String withPwd) {
    for (Customer cust:customers) {
        if (cust.nameMatches(custname)) {
            cust.tryLogin(withPwd);
        }
    }
    return "Oops -- don't know this customer";
}
```

### 3 Communicating History Between Methods

We noted that `withdraw` should require the account owner to be logged in. Logging in happens in the `login` method. This means that the two methods need to communicate about the history of calls to `login`. How do methods communicate in Java?

One way methods communicate is through passing parameters. But parameters don’t keep track of history. To track history, we need variables whose values are updated to store the history we will need later.

To address this here, we’ll add a list of logged in users to the `BankingService`. The `login` method will add to the list, and `withdraw` will consult it (using the built-in `contains` method on `LinkedLists`). (Shouldn’t we also have logout? Yes, we should, but that’s orthogonal to the current discussion.)

```java
class BankingService {

```
This example illustrates how fields (or variables) are used to manage history that must be shared or communicated between methods.

4 Encapsulating Representation

We did not cover this section in lecture. It is here for reference should you care to read the details

Now we return to the third problem we cited in our critique of the original code: the BankingService class fixes the assumption that accounts and customers should be stored as linked lists. If we have a lot of accounts, for example, we might want to be able to store them in an array. Right now, the methods in BankingService have been written specifically for LinkedLists (due to the use of the for-loops). Code designed for long-term evolution and maintenance (in other words, most code in a production environment) should NOT do this.

To fix this, we need to rewrite the code to remove both the iterator-for-loops and the specific references to LinkedList. But how? This involves several steps, described over the next several subsections.

4.1 Replace Fixed Data Structures with Interfaces

In general, here is how to factor a fixed data structure out of existing code:

1. Find all variables whose type you want to generalize.

2. Introduce interfaces for the types of these variables (some variables may be able to share the same types).
3. For each place in the current code that relies on the current type of the variable, ask yourself what that code is trying to compute (i.e., figure out a purpose statement for it). Invent a method name for that computation, add it to the interface, and replace the existing code with a call to the new method.

To make this clearer, let’s apply these steps to our BankingService program.

1. **Which variables to we want to generalize?**: Each of accounts and customers.

2. **Choose interface names for the variables**: Each of these is representing a set, so `IAccountSet` and `ICustSet` are reasonable choices.

   ```java
   interface IAccountSet {}
   interface ICustSet {}
   
   class BankingService {
     private IAccountSet accounts;
     private ICustSet customers;
     ...
   }
   ```

3. **For each place in the current code that relies on the current type of the variable, ask yourself what that code is trying to compute**: Let’s take the original `getBalance` code as an example.

   ```java
   double getBalance(int forAcctNum) {
     for (Account acct:accounts) {
       if (acct.numMatches(forAcctNum))
         return acct.getBalance();
     }
     return 0;
   }
   ```

   The for-loop here locates the account with the given number, then gets the balance from that account. The general purpose of the for-loop, then, is to find an account by its number. This suggests the following method on `IAccountSet`:

   ```java
   interface IAccountSet {
     // returns the account whose number matches the given number
     Account findByNumber(int givenNum);
   }
   ```

4. **Replace current code on a specific data type with calls to methods in the new, general, interface**: Now, we rewrite `getBalance` to use `findByNumber`.

   ```java
   double getBalance(int forAcctNum) {
     Account acct = findByNumber(forAcctNum);
     return acct.getBalance();
   }
   ```
Note that we have not yet addressed what happens if there is no account with the given number in the list. We will return to that in the next lecture.

Follow similar steps to generalize the withdraw and login methods. We leave these as an exercise so you can practice.

4.2 Create Concrete Classes that Implement the New Interfaces

Now that we have rewritten BankingService to use IAccountSet and ICustSet, we need classes that implement these interfaces. Our original code provides an initial implementation using LinkedList.

```java
class AcctSetList implements IAccountSet {
    LinkedList<Account> accounts;

    public Account findByNumber(int givenNum) {
        for (Account acct:accounts) {
            if (acct.numMatches(givenNum))
                return acct;
        }
        return null; // not good -- will fix when we handle errors
    }
}
```

With the generalized findByNumber method, it isn’t clear what to use as the return type if no account has the given number: different methods that call this search method might need different default answers. For now, we will use the very wrong approach of returning null, just so we can get the code to compile. We will discuss how to do this properly in a couple of weeks.

4.3 Initialize Data with Objects of the New Concrete Class

We have generalized BankingService and made new classes for the data structures we need. One step remains: we have to tell the BankingService to use our concrete classes. Where should this happen?

It should not happen within BankingService itself. The whole point of encapsulation is that BankingService shouldn’t know which specific data structures it is using. The only other way to get a specific object into a BankingService object is through the constructor. This is the answer: when you create a BankingService object, pass it objects of the specific data structure that you want to use.

```java
class Examples {
    BankingService B = new BankingService(new AcctSetList(),
        new CustSetList());

    ...
}
```

This illustrates how we create different banking services with different data structures for accounts and customers. If we had an Array-based implementation of IAccountSet as a class named AcctSetArray, we could create a different banking service using:
BankingService B = new BankingService(new AcctSetArray(),
new CustSetList());

Since BankingService only uses methods in the IAccountSet and ICustSet interfaces, we can freely chose a data structure without editing the code within the BankingService class (which was our goal).

4.4 Anticipated Questions

- *This banking service has no customers or accounts. How do we populate those?*
  
  In a full banking service program, the IAccountSet and ICustSet interfaces would also need methods for adding new elements. These notes do not include these in order to stay focused on the topic at hand.

- *If I have to specify the data structure to use in the constructor, how would I switch to a new data structure after my banking service had been running for a while?*
  
  These notes have not addressed this question. To do this, you would first need a method to convert your existing data from one representation to the other. Then, you would either create a new BankingService with the new data, or use some method provided in the BankingService to update the data structure.

  Something to think about: Providing a simple setter method would let you change the data structure. What’s wrong with this solution? What would a better solution look like?

5 Summary

Compare the original banking code to the version posted with today’s notes. The new BankingService is much cleaner and more maintainable. It allows the information about accounts and customers to change with less impact on the banking service methods. The banking service no longer relies on any particular data structure for accounts and customers. We achieved both of these goals by isolating data and methods in classes, and using interfaces to separate general data from implementation details.

Key take-aways from these lectures:

- Encapsulation is about putting data and the methods that operate on that data together. OO classes provide a natural mechanism for doing this.

- Encapsulation matters because it lets you change data and how it is used without editing existing code. This lecture has shown two examples of this:
  
  - We might add information (such as a withdrawal fee) and want to change methods (such as withdraw) to use that information.
  
  - Writing code that can be customized to different specific data structures (such as linked lists versus arrays).

- Java provides access modifiers that let you control which other classes can access your methods and fields. You should put explicit access modifiers on all of your fields and methods.
Encapsulation is NOT just about protecting your data. It is about your class hierarchy (more generally, the architecture of your program). The Java access modifiers are used in conjunction with encapsulation (and indeed help reinforce encapsulation), but encapsulation is a much broader topic.

Encapsulation is an important issue no matter what language you are programming in. Different languages provide different support for encapsulation. Java’s support comes in the form of classes and interfaces (supported by access modifiers). Other languages have other mechanisms. When designing a new product in any language, it is important to ask what information you want to protect and what decisions you want to be able to change later, then understand how the language can help you achieve those goals.

5.1 Two Myths About Encapsulation

Those of you with prior Java experience may have heard two general guidelines or slogans that aren’t quite accurate:

- **MYTH: Always make a getter and setter for every private field.** No. Sometimes, we have fields that we don’t want anyone to have access to – they are for internal use only. Creating getters/setters for such fields contradicts the design goal of those fields. Would you publish a getter for the password field of Customer, for example? No – that data should only be handled within the Customer.

  This rule can also violate representation encapsulation. Imagine that ICustSet provided a method getCustomers that returned the customers field. That would defeat the whole purpose of hiding the representation, because a programmer could get the actual data structure and write code against it (like a for-loop against the LinkedList). This is a great example of where the getter is exactly the wrong thing to provide.

- **MYTH: Encapsulation equals Information Hiding.** These concepts are related, but not equal. One can encapsulate data and methods, but still not hide information (if you expose everything through getters, for example). Encapsulation is part of a solution for hiding information, but information hiding needs careful design beyond just putting all data and methods together in a class.

5.2 The Many Faces of Variables

At this point, it is worth stepping back to review the many uses we have seen for variables in the course so far:

- To capture fields whose values might change over time (like the balance field in Accounts)
- To track the progress of a local computation within a method, (like the pos variable used to track the steps of a for-loop); this includes storing the result of part of a computation so you can return it at the end of a method
- To capture the history of one computation (like logging in) for use by another (like withdraw)
Why do we point out these different uses? Because conceptually they are very different, but notationally they all look the same in Java. When you are working on a problem, it’s worth thinking about which of these uses might arise in your program so you can create and manage variables appropriately.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/cs018/feedback