Lecture 12: Access Modifiers and Encapsulation

10:00 AM, Feb 21, 2020

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

Motivating Question

How do we limit the ability of classes to read and write fields of other classes in Java?

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

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 how to restrict access to data stored in Java fields.

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);
        }
    }
    return 0;
}
```

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.
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
```
```java
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";
}
```

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: [https://cs.brown.edu/courses/cs018/feedback](https://cs.brown.edu/courses/cs018/feedback)