Lecture 13: Access Modifiers and Encapsulation

11:00 AM, Feb 23, 2020

Contents

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?

https://brown-cs18-master.github.io/content/lectures/13accessEncap/single-page-starter.pdf

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. Even the `main` method can change the password. Such changes should not be permitted.

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 for lookups, 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 Protecting Data Access

Let's start by protecting the customer's password. We want to prevent arbitrary code from accessing or modifying the password, We do this by adding the annotation `private` to the line that defines `password`, as follows:

```
public class Customer {
  private String password;
  ...
}
```

When we do this, the use of `password` in the `login` method (in `BankingService` breaks. The `private` annotation prevents any use of the `password` field outside of the `Customer` class. To fix

this, we have to replace the direct use of `password` with a call to a method within the `Customer` class that does the same work. This could look as follows:

```
public class Customer {
   String name;
   private String password;  // don't allow read/write outside this class
   LinkedList<Account> accounts = new LinkedList<Account>();

   public boolean checkPwd(String checkAgainst) {
       return this.password.equals(checkAgainst);
   }
}

public class BankingService {
   public String login(String custname, String withPwd) {
        for (Customer cust:customers) {
            if (cust.name.equals(custname)) {
              if (cust.checkPwd(withPwd)) { // this line changed
                ...
              }
        ...
}
```

We might also have handled password access by adding the following method to the `Customer` class:

```
public String getPassword() {
   return this.password;
}
```

which would have allowed the following version of the **`if`** statement in the `login` method:

```
   if (cust.getPassword.equals(withPwd))
```

Stop and Think: Is this solution also acceptable?

This is a terrible idea! We've just handed out the password to anyone who asks for it (thus defeating the point of keeping the password private). Methods like `getPassword` are commonly called *getters*. Some OO coding guides recommend creating getters and setters for all of your fields. But this is really bad practice. It is MUCH better to create methods like `checkPwd` that provide exactly what operations you want to allow on data, rather than hand out data and let other programmers decide what to do with them.

There are isolated cases in which getters (and their analogs, *setters*, for updating data values) make sense. But you should only create one of these when there is a good reason to do so and a more specialized method would not be appropriate.

2.1 Summary of Java Access Modifiers

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.

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*.

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.

2.1.1 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.

3 Encapsulation

Using access modifiers forces us to follow a practice known as *encapsulation*, in which methods are placed in the same classes as the data that they operate on. Encapsulation is good object-oriented design, whether or not you are working with data that needs to be protected. Why? Because if you then have to change data later, the methods that might be affected are in the same class.

For example, imagine that the bank decides to add limits on the number of allowed withdrawals made per month. This would require a new field, which would have to be consulted when making a withdrawal. If all `withdraw`-style methods are in the `Account` class, then it is easy for someone to find the affected methods and change them. But if the core code for processing a withdrawal is in a different class, then someone has to read all of the code that uses accounts to see whether any are affected by the new field. In a large project with hundreds of thousands of lines of code, this would be an organizational disaster. Following good encapsulation practices, which would have all methods related to account data in the `Account` class, keeps this task manageable.

3.1 Putting Methods in Their Proper Place

In the spirit of encapsulation, let's move the `withdraw` and `getBalance` methods into the `Account` class:

```
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;
  }
}
```

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`.

```
  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;
  }
```

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

```
public class Customer {
  String name;
  private 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";
  }
}
```

3.2 More Access Modifiers in the Banking Service

Set the `balance` field in the `Account` class to be private. What issues did the compiler flag? Only the one in the `Main` class, unlike when we made the `password` private. Why did this go more smoothly? Because by practicing good encapsulation (moving `withdraw` to the `Account` class), our code was already set up to respect the privacy of the `balance` field. This shows how data protection and encapsulation go hand in hand.

Make the rest of the `Customer` and `Account` fields private, then think about how to adapt the code. 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).

Try it: Put a `nameMatches` method in `Customer` that takes the name to compare to and checks whether the names are equal.

Our `Customer` class should now look at follows:

```
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));
  }

  // check whether given password matches this password
  public boolean checkPwd(String checkAgainst) {
       return this.password.equals(checkAgainst);
   }

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

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

```
  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";
  }
```

3.3 Private Methods?

The `tryLogin` method was the only one using the `checkPwd` method, which we created earlier. We could fold `checkPwd` into `tryLogin`, but we'll leave it here to illustrate a different point: `checkPwd` is now just a helper function within the `Customer` class. As such, we can prevent any other class from using the method by marking it `private`:

```
// check whether given password matches this password
private boolean checkPwd(String checkAgainst) {
     return this.password.equals(checkAgainst);
}
```

This is a good rule of thumb in general: make methods private unless some other class needs to use them.

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`.