CS200: Model-View-Controller and Exceptions

Kathi Fisler

February 25, 2022

Motivating Questions

- How should the many components of a program with a user-interface be split across multiple classes?
- How to we handle errors without terminating the entire program?

1 Motivating Context

Our last version of the banking example (see website for starter code) doesn't handle login errors well. It either accepts the login or crashes the system with a null pointer exception. Ideally, we need something more nuanced – if someone enters an invalid username or password, we would ideally prompt them to log in again.

2 A Simple Banking Application

Figure 1 (next page) gives the essential code for a simple banking application (we used parts of this in the last lecture on debugging plans). The application lets customers withdraw and deposit funds, and also has a notion of customers logging in. It also has a simple text-based interface for letting a user communicate with the program:

We could start the bank application with the following code:

```
public static void main(String[] args) {
        BankingService B = new BankingService();
        Customer kCust = new Customer("kathi", "cs18");
        Account kAcct = new Account(100465, kCust, 150);
        B.addAccount(kAcct);
        B.loginScreen();
        kAcct.printBalance();
        B.withdraw(100465, 30);
        kAcct.printBalance();
}
```

The `BankingService` class contains the core data structures and methods for banking operations, along with methods for letting a user interact with those methods. Right now, all of this code is in one big class. But when designing an application, we should ask ourselves the following question:

Might I want to swap out different implementations of any of these parts later?

Classes and objects are a nice structuring method because you could use different classes to provide the same functionality (as we saw with `LinkedList` versus `ArrayList`), yet with different implementation decisions and performance. So let's critique our current `BankingService` by asking whether any parts of it might be worth swapping out with other detailed code later.

Stop and Think: What might you swap out?

```java
public class Customer {
  private String name;
  private String password;
}

class BankingService {
  LinkedList<Customer> customers = new LinkedList<Customer>();

  // return the Customer whose name matches the given name
  public Customer findCustomer(String custname) {
    for (Customer cust : customers) {
      if (cust.nameMatches(custname))
      return cust;
    }
    return null;
  }

  public String login(String custname, String pwd) {
    Customer cust = findCustomer(custname);
    if (cust.checkPwd(pwd)) {
      return "Welcome";
    } else {
      return "Try␣again";
    }
  }

  public void loginScreen() {
    // Set up an object that reads text from the keyboard
    // (you will learn about this in I/O lab)
    Scanner keyboard = new Scanner(System.in);
    System.out.println("Welcome␣to␣the␣Bank!␣Please␣log␣in.");
    System.out.println("Enter␣username:␣");
    String username = keyboard.next();
    System.out.println("Enter␣password:␣");
    String password = keyboard.next();
    this.login(username, password);
    System.out.println("Thanks␣for␣logging␣in!");
  }
}
```

Figure 1: The initial Banking code

Two specific aspects of this code stand out as being worthy of swapping out:

1. Which data structure we use for customers
2. What kind of interface we provide (website, keyboard I/O, voice-driven, etc)

Our task today is to restructure the code to handle the second. We'll return to the first next week.

2.1 Pulling out the User Interface

To pull out the user interface, we make a new class for the interface code. Here, we're calling it `BankingConsole`. We'll move the `loginScreen` method over to this new class:

```java
import java.util.Scanner;

public class BankingConsole {
  private Scanner keyboard = new Scanner(System.in);

  public void loginScreen() {
    System.out.println("Welcome to the Bank.  Please log in.");
    System.out.print("Enter your username: ");
    String username = keyboard.next();
    System.out.print("Enter your password: ");
    String password = keyboard.next();
    this.login(username, password);
    System.out.println("Thanks for logging in!");
  }
}
```

IntelliJ flags the call to `this.login` as having an error. Now that we've moved the `loginScreen` out of `BankingService`, the `login` method is no longer in the `this` class. Removing `this` doesn't help: the `login` method is still in the `BankingServices` class. So what do we do? Two suggestions jump to mind:

1. move the `login` method from `BankingService` into `BankingConsole` as well
2. have the `loginScreen` method take a `BankingService` object as input (which we could then use to access the `login` method)

The first option ends up not making sense: the `login` method isn't really about the interface, so it doesn't seem to belong in the class for the interface code. More practically, if we move `login`, we'd then run into a similar problem with `findCustomer`, and that definitely isn't related to the user interface. So perhaps we should try the second option.

The second option would work. But we actually want to solve this problem slightly differently. It will turn out that many `BankingConsole` methods would need to access methods in `BankingService` (imagine that the user interface gave someone a way to make a withdrawal, for example). Rather than have all of them take a `BankingService` object as input, we can pass a single `BankingService` object to the `BankingConsole` constructor, then use that for all service operations. The code would look as follows:

```java
import java.util.Scanner;

public class BankingConsole {
  private Scanner keyboard = new Scanner(System.in);
  private BankingService forService;

  public BankingConsole(BankingService forService) {
    this.forService = forService;
  }

  public void loginScreen() {
    System.out.println("Welcome␣to␣the␣Bank.␣␣Please␣log␣in.");
    System.out.print("Enter␣your␣username:␣");
    String username = keyboard.next();
    System.out.print("Enter␣your␣password:␣");
    String password = keyboard.next();
    forService.login(username, password);
    System.out.println("Thanks␣for␣logging␣in!");
  }
}
```

3 Model-View-Controller

With the addition of the `BankingConsole`, we can talk about the overall architecture (configuration and roles) of the application and its classes. We talked about how the classes can be divided into three roles, as shown in the diagram in lecture.

- The **view** (`BankingConsole`), which contains the code that interacts with the user (whether text I/O, audio, web interface, etc). The user gives input to the view, which executes commands in the application through the ...
- **controller** (`BankingService`), which contains methods for the major operations that the application provides (like logging in, withdrawing funds, etc). Once the controller knows what the user wants to do, it coordinates actual completion of a task by calling methods in the ...
- **model** (`Customer`), classes that contain the data and perform operations on the data to fulfill application tasks.

This architecture, known as *model-view-controller* is quite common in software engineering. It reinforces the idea that the interface code should be separate from the underlying operations, and that the underlying operations should be expressible against a variety of data structures. The details of the data structures live in their own classes, with fields protected through access modifiers. This enables updating an application with different data details without having to reimplement the core logic.

4 Handling Failed Login Attempts

In the original code, if a login attempt fails, the system prints that login has failed then thanks the user for logging in. This is obviously broken. Ideally, we should check whether the login has been successful and prompt the user to try again if it was not. Let's work through some ways to try doing that:

4.1 Relogin Poorly with If-statements

What might a revised `loginScreen` method look like if we re-prompted login using if-statements?

```
public class BankingConsole {
  public void loginScreen() {
    Scanner keyboard = new Scanner(System.in);
    System.out.println("Welcome␣to␣the␣Bank.␣Please␣log␣in.");
    System.out.print("Enter␣your␣username:␣");
    String username = keyboard.next();
    System.out.print("Enter␣your␣password:␣");
    String password = keyboard.next();
```

```
      if (controller.login(username, password))
         System.out.println("Thanks␣for␣logging␣in!");
      else {
        System.out.println("Login␣failed.␣Please␣try␣again");
        this.loginScreen();
      }
    }
  }
```

This would require the `login` method to change to return a boolean, which would in turn require checking whether `findCustomer` had returned **`null`**.

```
public boolean login(String custname, String withPwd) {
  Customer cust = customers.findCustomer(custname);
  if (cust == null)
    return false;
  else if (cust.checkPwd(withPwd)) {
    this.loggedIn.addFirst(cust);
    return true;
  } else {
    return false;
  }
}
```

Notice how the code is getting a bit cluttered with the nested **`if`** statements? It would be much nicer to have the `login` method be clean and streamlined, focusing on the core logic of logging in. Something like:

```
public void login(String custname, String withPwd) {
  Customer cust = customers.findCustomer(custname);
  if (cust.checkPwd(withPwd)) {
    this.loggedIn.addFirst(cust);
  }
}
```

5 Exceptions

Right now, `findCustomer` returns **`null`**, as a way of saying ”no answer”. While this approach gets past the compiler, it is not a good solution because it clutters up the code of other methods that call `findCustomer`. The better approach is to raise an error that our program can handle (rather than crashing, as a `RuntimeException` would. Exceptions are designed to help programs flag and handle situations that would otherwise complicate the normal logic of the program you are trying to write.

Exceptions (or some similar notion) exist in most mainstream programming languages. Intuitively, if a function encounters a situation that is not expected, it does not try to return a normal value. Instead, it announces that a problem occured (by *throwing* or *raising* an exception). Other methods watch for announcements and try to recover gracefully.

5.1 Creating and Throwing Exceptions

Our goal is to replace the **`return null`** statement from the current `findCustomer` code with an exception to alert to the rest of the code that something unexpected happened (in this case, the customer was not found). The Java construct that raises alerts is called **`throw`**. Our first step, then, is to replace the **`return null`** statement with a **`throw`** statement:

```
public Customer findCustomer(String custname) {
   for (Customer cust : customers) {
```

```java
    if (cust.nameMatches(custname)) {
      return cust;
    }
  }
  // replace return null with a report of an error
  throw new CustomerNotFoundException(custname);
}
```

CustomerNotFoundException is a new class that we will create to hold information relevant to the error that occured (in this case, which customer couldn't be found – we might want to print this information out as part of error message later). We create a subclass of Exception for each different type of alert that we want to raise in our program.

```java
class CustNotFoundException extends Exception {
  String unfoundName;

  CustNotFoundException(String name) {
    this.unfoundName = name;
  }
}
```

An exception subclass should store any information that might be needed later to respond to the exception. In this case, we store the name of the customer that could not be found. This info would be useful, for example, in printing an error message that indicated which specific customer name could not be found.

Summarizing: we modify findCustomer to throw a CustNotFoundException if it fails to find the customer. Three modifications are required:

- The **throw** statement needs to be given an object of the CustNotFoundException class to throw.
- The findCustomer method must declare that it can throw that exception (the compiler needs this information). This occurs in a new **throws** declaration within the method header, as shown below.
- The ICustomerSet interface, which has the findCustomer method header, must also include the **throws** statement.

```java
interface ICustomerSet {
  Customer findCustomer(String name) throws CustNotFoundException;
}

class CustomerList implements ICustomerSet {
  ...
  // return the Customer whose name matches the given name
  public Customer findCustomer(String custname)
     throws CustomerNotFoundException {
    for (Customer cust : customers) {
      if (cust.nameMatches(custname)) {
        return cust;
      }
    }
    // replace return null with a report of an error
    throw new CustomerNotFoundException(custname);
  }
}
```

5.2 Catching Exceptions

Exceptions are neat because they let us (as programmers) control what part of the code handles the errors that exceptions report. Think about what happens when you encounter a login error when using a modern web-based application: the webpage (your user interface) tells you that your username or password was incorrect and prompts you to try logging in again. That's the same behavior we want to implement here.

To do this at the level of code, we will use another new construct in Java called a **try**-**catch** block. We "try" running some method that might result in an exception. If the exception is thrown, we "catch" it and handle it. Here's a **try**-**catch** pair within the `loginScreen` (which is where we already said we want to handle the error:

```java
public void loginScreen() {
  Scanner keyboard = new Scanner(System.in);
  System.out.println("Welcome␣to␣the␣Bank.␣␣Please␣log␣in.");
  System.out.print("Enter␣your␣username:␣");
  String username = keyboard.next();
  System.out.print("Enter␣your␣password:␣");
  String password = keyboard.next();
  try {
    controller.login(username, password);
    System.out.println("Thanks␣for␣logging␣in!");
  } catch (CustomerNotFoundException e) {
    // what to do when this happens
    System.out.println("No␣user␣" + e.custname);
    this.loginScreen();
  }
}
```

Notice the **try** encloses both the call to `login` and the `println` that login succeeded. When you set up a try, you have it enclose the entire sequence of statements that should happen if the exception does NOT get thrown. As Java runs your program, if any statement in the **try** block yields an exception, Java ignores the rest of the **try** block and hops down to the **catch** block. Java runs the code in the **catch** block, and continues from there.

The `e` after `CustomerNotFoundException` in the **catch** line refers to the exception object that got thrown. As the code shows, we could look inside that object to retrieve information that is useful for handling the error (like printing an error message).

Note: if you've only typed in the code to this point and try to compile, you will get errors regarding the `login` method – hang on – we're getting to those by way of the next section.

5.3 Understanding Exceptions by Understanding Call Stacks

To understand how exceptions work, you need to understand a bit more about how Java evaluates your programs.

Exceptions aside, what happens "under the hood" when Java runs your program and someone tries to log in? Our `main` method started by calling the `loginScreen` method; this method calls other methods in turn, with methods often waiting on the results of other methods to continue their own computations. Java maintains a *stack* (we discussed those briefly in the data structures lectures) of method calls that haven't yet completed. When we kick off `loginScreen`, this stack contains just the call to that method.

Separately from the stack, Java starts running the code in your method statement by statement. Switch now to the PDF linked next to these notes on the lectures page ("How Exeptions Work"), which walks through how Java executes programs with **try**/**catch** blocks, showing how the exceptions work. The slideshow simplifies a couple of details. There may be multiple **try** markers on the stack (because you can have multiple **try** blocks), and the stack has ways of "remembering" where it left off in pending method calls. We ignore those details here in the hopes of giving you the bigger picture.

5.4 Housekeeping: annotating intermediate methods

As our demonstration of the stack just showed, the `CustomerNotFoundException` ”passes through” certain classes as it comes back from the `findCustomer` method. The Java compiler needs every method to acknowledge what exceptions might get thrown while it is running. We therefore have to add the same **`throws`** annotations to each method that does not want to catch the exception as it passes through on the way to the marker. For example, the `login` method needs to look as follows:

```
public String login(String custname, String withPwd)
            throws CustomerNotFoundException, LoginFailedException {
    Customer cust = customers.findCustomer(custname);
    if (cust.checkPwd(withPwd)) {
      System.out.println(’’Login Successful’’);
      this.loggedIn.addFirst(cust);
    } else {
       System.out.println(’’Login Failed);
     }
}
```

Once you put these additional **`throws`** annotations on the code, the code should compile and Java will report failed logins through the loginScreen.

6 Summarizing `Try/Catch` blocks

At this point, you should understand that **`throw`** statements go hand-in-hand with **`try`**/**`catch`** blocks. Whenever a method declares that it can throw an exception, any method that calls it needs a **`try`**/**`catch`** statement to process the exception.

More generally, a **`try`**-**`catch`** block looks as follows:

```
try {
  <the code to run, assuming no exceptions>
} catch <Exception> {
  <how to recover from the exception>
}
```

You can have multiple catch phrases, one for each kind of exception that you need to handle differently (or you can have two kinds of exceptions get handled the same way, as we will show shortly).

6.1 Handling Incorrect Passwords

Now that you’ve seen one example of exceptions, let’s try another. As an exercise for yourself, change the `login` method in the `CustomerList` class (which checks the password) so that it throws an exception called `LoginFailedException` if the passwords don’t match.

Try it before reading further.

You should have ended up with the following:

```
public String login(String custname, String withPwd)
            throws CustomerNotFoundException, LoginFailedException {
    Customer cust = customers.findCustomer(custname);
    if (!cust.checkPwd(withPwd))
       throw new LoginFailedException(custname);
    this.loggedIn.addFirst(cust);
}
```

In addition, we have to add this exception to the **`catch`** used to prompt a user to log in again.

```java
public void loginScreen() {
  Scanner keyboard = new Scanner(System.in);
  System.out.println("Welcome␣to␣the␣Bank.␣␣Please␣log␣in.");
  System.out.print("Enter␣your␣username:␣");
  String username = keyboard.next();
  System.out.print("Enter␣your␣password:␣");
  String password = keyboard.next();
  try {
    controller.login(username, password);
    System.out.println("Thanks␣for␣logging␣in!");
  } catch (CustomerNotFoundException|LoginFailedException e) {
    // what to do when this happens
    System.out.println("Login␣Failed'');
␣␣␣␣this.loginScreen();
␣␣}
}
```

As with the `CustNotFoundException`, you have to put **`throws`** annotations on all methods that can either throw or pass along the `FailedLoginException`. You'll see this in the final posted code.

7 Checked Versus Unchecked Exceptions

What we have done so far with **`try`**/**`catch`** and **`throw`** statements are called *Checked Exceptions*: exceptions that you are using within your application to respond to special situations that arise within your code. With checked exceptions, Java analyzes your code at compile time to make sure that the exceptions will actually be caught (and handled).

Sometimes, however, your code fails because of a bug in your code. Null pointer exceptions, array index out of bounds exceptions, and division by zero exceptions are examples of exceptions that alert you to bugs when they are thrown. You don't want to catch these – you want to fix your code so that they can't happen again. Put differently, the fix for such situations is to debug your code before you run it. Checked exceptions, in contrast, are for situations you can't control (because they arise from user behavior, for example) that arise while your application is running.

These "code-error" exceptions are special cases of `RuntimeExceptions`, which we saw earlier in the course. When you throw a runtime exception, you shouldn't catch and manage it (and the compiler won't expect you to). Runtime exceptions are handy when you are working on your code and just want to throw something so you can get past the compiler and test the code you're currently working on. In production code, you only use them for situations in which the program needs to terminate.

As a general rule, you should write (and catch) checked exceptions from here on out in homeworks and projects.