Contents

1 Exceptions and How They Work 1
  1.1 Update to the Banking Example 1

2 Back to the Bank 2

3 What is an Exception? 3
  3.1 Creating and Throwing Exceptions 3
  3.2 Catching Exceptions 4
  3.3 Understanding Exceptions by Understanding Call Stacks 5
  3.4 Housekeeping: annotating intermediate methods 6

4 Summarizing Try/Catch blocks 6
  4.1 Handling Incorrect Passwords 6
  4.2 Optional: Converting One Exception To Another 7

5 Checked Versus Unchecked Exceptions 8

Objectives

By the end of these notes, you will know:

- when to create and manage your own exceptions
- how exceptions are actually processed when programs run

By the end of these notes, you will be able to:

- create your own exceptions
- use try, catch, and throws statements to manage your own exceptions
- understand which lines of code do and don’t get executed when exceptions are thrown
1 Exceptions and How They Work

Recall the banking example from two weeks ago (when we studied about public and private annotations). At the time, we noted that we were just returning strings rather than reporting errors when people provided an incorrect password. By now, you’ve worked with exceptions enough to hopefully have guessed that an exception should be thrown here. We haven’t studied exceptions formally though. In this lecture, we’re going to look carefully at exceptions and how they get evaluated within programs.

1.1 Update to the Banking Example

The last time we looked at the banking example, we had the following data organization:

- classes for Customers and Accounts
- a BankingService containing lists of customers and accounts

The version we are starting from today has two main extensions:

- We added a BankingConsole class with a simple Scanner-based interface for interacting with the bank
- We put the linked lists of customers and accounts inside separate classes, to which we could add methods for searching for items, adding items, and so forth. Creating a separate class like this gives us a way to combine methods specific to our application with built-in Java classes to represent the data.

Here is the link to the starter file:
http://cs.brown.edu/courses/csci0180/content/sources/exceptions-starter.java

2 Back to the Bank

Here’s our core question: what should the findByNum and findByName methods do if no customer or account matched the given input data?

Right now, both methods return 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 findByNum and findByName. Think about getBalance. Ideally, we would like to write this method as follows:

```
class BankingService {
    ...
    public double getBalance(int forAcctNum) {
        Account acct = accounts.findByNum(forAcctNum);
        return acct.getBalance();
    }
}
```
However, if `findByNum` returns `null` (to indicate that no such account number exists), Java will raise an error at runtime when it tries to compute `acct.getBalance()`. To guard against that, we might modify the code to read:

```java
class BankingService {
   ...
   public double getBalance(int forAcctNum) {
      Account acct = accounts.findByNum(forAcctNum);
      if (acct != null)
         return acct.getBalance();
      else
         return ???; // some dummy value must go here
   }
}
```

This code has two key flaws:

1. The logic of the program has been obscured by the `if`-statement. The `if`-statement is really a check on the behavior of a different method (`findByNum`); it should not clutter up the code of this one.

2. Returning dummy values is never a good idea, because the computation that receives the dummy value must be able to distinguish valid from invalid data. That requires something akin to the `if`-statement, which we argued against in the previous point.

Ideally, we need a way to clearly write methods that use `findByNum` and `findByName`, while letting these two methods alert methods that call them when something goes wrong. The appropriate programming construct for this is called an exception. As the name implies, 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.

### 3 What is an Exception?

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 occurred (by throwing or raising an exception). Other methods watch for announcements and try to recover gracefully.

Isn’t returning `null` an announcement that something went wrong? Yes, but we’ve just seen that if we return `null`, the code that called `findByNum` has to check whether an announcement got made and handle it before calling `getBalance`. Exceptions use a separate communication channel (as it were) for announcements. This lets methods separate out the "normal" code from the "announcement-handling" code; it keeps the code cleaner and will help direct announcements to the part of the code that can best respond to them.

We’ll make this concept more concrete by working with `findByName`. 
3.1 Creating and Throwing Exceptions

Our goal is to replace the return null statement from the current findByName 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:

```java
class CustSet implements ICustSet {
    ...
    // return the Customer whose name matches the given name
    public Customer findByName(String name) {
        for (Customer cust:customers) {
            if (cust.getName() == name)
                return cust;
        }
        throw <some object indicating that the name was not found>;
    }
}
```

Next, we need to provide the specific exception to throw. In Java, an exception is an object in the Exception class. We create a subclass of Exception for each different type of alert that we want to raise in our program. In this case, we will create a new exception class for user-not-found errors:

```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.

Finally, we modify findByName 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 findByName 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 ICustSet interface, which has the findByName method header, must also include the throws statement.
interface ICustSet {
    Customer findByName(String name) throws CustNotFoundException;
}

class CustSet implements ICustSet {
    ...
    // return the Customer whose name matches the given name
    public Customer findByName(String name) throws CustNotFoundException {
        for (Customer cust:customers) {
            if (cust.getName() == name)
                return cust;
        }
        throw new CustNotFoundException(name);
    }
}

3.2 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:

```
public void loginScreen() {
    // prompt for user to enter name and password
    System.out.println("Welcome to the Bank. Please log in.");
    String username = keyboard.next();
    int password = keyboard.nextInt();
    try {
        B.login(username, password);
        System.out.println("Login successful");
    } catch (CustNotFoundException e) {
        System.out.println("Login Failed. Try Again");
        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.
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.]

### 3.3 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. Imagine an arrow alongside your code that tracks which statement Java is currently evaluating (statements above the arrow are already completed).

Switch now to the following PDF, which walks through how Java executes programs with try/catch blocks, showing what we mean by the "arrow alongside code".

http://cs.brown.edu/courses/csci0180/content/lectures/aux/ExceptionControlFlow.pdf

The slideshow you just saw 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.

### 3.4 Housekeeping: annotating intermediate methods

As our demonstration of the stack just showed, the CustNotFoundException exception "passes through" certain classes as it comes back from the findByName 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:

```java
public void login(String custname, int withPwd) throws CustNotFoundException {
    Customer cust = customers.findByName(custname);
    cust.tryLogin(withPwd);
}
```

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

### 4 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:

```java
try {
    <the code to run, assuming no exceptions>
} catch <Exception> { 
    <how to recover from the exception>
}
```
You can have multiple catch phrases, as we will see in later examples.

### 4.1 Handling Incorrect Passwords

Now that you’ve seen one example of exceptions, let’s try another. As an exercise for yourself, change the tryLogin method in the Customer class (which checks the password) so that it throws an exception from a new class called LoginFailedException.

Try it before reading further.

You should have ended up with the following:

```java
// ---- the new exception class ----
class LoginFailedException extends Exception {
    LoginFailedException() {
    }
}
// ---- in the customer class ----
class Customer {
    // check whether the given password matches the one for this user
    public void tryLogin(int withPwd) throws LoginFailedException {
        if (this.password != withPwd)
            throw new LoginFailedException();
    }
}
```

In addition, we have to add a catch block for these exceptions. Where do we want to catch a FailedLoginException? As before, we would like to catch this in the loginScreen method. We can add another catch block to that method.

```java
public void loginScreen() {
    // prompt for user to enter name and password
    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: ");
    int password = keyboard.nextInt();
    try {
        B.login(username, password);
        System.out.println("Login successful");
    } catch (CustNotFoundException e) {
        System.out.println("Login Failed. Try Again");
        this.loginScreen();
    } catch (FailedLoginException e) {
        System.out.println("Login Failed. Try Again");
        this.loginScreen();
    }
}
```

As with the CustNotFoundException, you have to put throws annotations on all methods that can either throw or pass along the FailedLoginException.

### 4.2 Optional: Converting One Exception To Another

Some of you may look at the catch clauses in the loginScreen method and have some concerns: Both exceptions seem to point to the same core problem (with the same recovery) – we couldn’t log in the user. Indeed, many systems don’t tell the
user whether their login problem was due to the username or the password. It’s all just one common “failed login” message and recovery.

What if we wanted the loginScreen to only get FailedLoginException, letting the login method somehow "convert" the CustomerNotFoundException to a FailedLoginException? There’s a nice design idea here – many methods may call findByName, but the significance of a failure to find the user is different in each case. Each such method could do a little processing on the exception to give its callers a more useful interpretation of the problem.

Let’s do that here: we will have the login method convert the CustomerNotFoundException to a FailedLoginException. This happens within the catch statement.

```java
class BankingService {
    ...
    public Customer login(String custname, int withPwd) throws LoginFailedException {
        try {
            Customer cust = customers.findByName(custname);
            cust.tryLogin(withPwd);
            return cust;
        } catch (CustomerNotFoundException e) {
            throw new LoginFailedException();
        }
    }
}
```

Now that this is done, the loginScreen will only ever catch FailedLoginExceptions.

```java
// set up a Scanner to read input from the keyboard
private Scanner keyboard = new Scanner(System.in);

// the method that prompts for input then tries to log 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: ");
    int password = keyboard.nextInt();
    try {
        this.login(username,password);
    } catch (LoginFailedException e) {
        System.out.println("Login failed -- please try again\n");
        this.loginScreen();
    }
}
```

We also must go through and fix the throws annotations on each method correspondingly.

The final code is available on the notes page.

5 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.

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/cs18/feedback](http://cs.brown.edu/courses/cs18/feedback).