7.5 Chained Exceptions

So far, we have only handled exceptions within the catch block in such a manner that the exception "disappears." No new exceptions are created within the exception handling code of the catch blocks. Depending on the situation, it may be useful to instead "transform" one type of exception into another within the catch block and throw the new type of exception for a higher level method to handle. That is, **the catch blocks do not have to "swallow up" their incoming exceptions; they can also throw exceptions**. When one exception causes another exception object to be created and thrown, we call this process a **chained exception**. In a domino effect, one exception can be thrown in Method A; caught, transformed into a new exception, and thrown again in Method B; caught, transformed into another exception, and thrown in Method C; and so on, until the "top-level" method finally handles the exception and "swallows it up."

We have revised our first example (**JavaExceptionEx1**) to use chained exceptions. The resulting code, **JavaExceptionEx3**, is presented below.

JavaExceptionEx3

Goal: Use chained exceptions

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class JavaExceptionEx3 {

    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        try {
            getInputsAndDisplay(input);
        } catch (IllegalArgumentException e) {
            System.out.println(e.getMessage());
        } finally {
            System.out.println("Ready to do another task in main()");
        }
    }
```

A1

```
    public static void getInputsAndDisplay(Scanner input)
        throws IllegalArgumentException {

        try {
            System.out.println("Enter two integers for division: A/B");
            System.out.println("A = ");
            // could throw InputMismatchException, or
            // NullPointerException if input is null
            int a = input.nextInt();
            System.out.println("B = ");
            // could throw InputMismatchException, or
            int b = input.nextInt();
            System.out.println(a + " / " + b + " is " + doDivision(a, b));
```

B1

```
        } catch (ArithmeticException e) {
            // from the doDivision invocation
            throw new IllegalArgumentException("ArithmeticException: "
                    + "Divisor cannot be zero ");
        } catch (InputMismatchException e) {
            throw new IllegalArgumentException("InputMismatchException: "
                    + "Inputs must be an integer");
        } catch (NullPointerException e) {
            throw new IllegalArgumentException("NullPointerException: "
                    + "Scanner input cannot be null");
        } finally {
            System.out.println("Finished getInputsAndDisplay()");
        }
    }
```

B2

```
    public static int doDivision(int a, int b) throws ArithmeticException {
            return a/b;
    }
}
```

C1

First, check out **Cell C1**:

```
public static int doDivision(int a, int b) throws ArithmeticException {
    return a/b;
}
```

We have defined a new method called `doDivision()` that performs simple integer division of two integer parameters. The method will throw `ArithmeticException` if the divisor is zero, as we have seen. Although `ArithmeticException` is *not* a checked exception, we have chosen to declare the exception in the method header to make it clear what exception the method could potentially throw.

We will invoke the `doDivision()` method in **Cell B1**:

```
public static void getInputsAndDisplay(Scanner input)
    throws IllegalArgumentException {

    try {
        System.out.println("Enter two integers for division: A/B");
        System.out.println("A = ");
        int a = input.nextInt();
        System.out.println("B = ");
        int b = input.nextInt();
        System.out.println(a + " / " + b + " is " + doDivision(a, b));
```

We have revised the original `doDivisionAndDisplay()` method into a method called `getInputsAndDisplay()`. This method will prompt the user for integer inputs, as before. In the parameter input is `null`, the invocation to `input.nextInt()` will throw a `NullPointerException` when the program attempts to save the variable `a`. If the user does not input a valid integer, the program will throw an `InputMismatchException`. Finally, if the user had inputted zero for the variable `b`, the invocation to `doDivision()` will throw an `ArithmeticException`, as specified in the method header for `doDivision()`.

Keeping these potential exceptions in mind, we handle them in the catch blocks that follow in **Cell B2**:

```
    } catch (ArithmeticException e) {
        throw new IllegalArgumentException("ArithmeticException: "
                + "Divisor cannot be zero ");
    } catch (InputMismatchException e) {
        throw new IllegalArgumentException("InputMismatchException: "
                + "Inputs must be an integer");
    } catch (NullPointerException e) {
        throw new IllegalArgumentException("NullPointerException: "
                + "Scanner input cannot be null");
    } finally {
        System.out.println("Finished getInputsAndDisplay()");
    }
}
```

In all three possible exception cases, the program will catch the generated exception and create a chained exception by re-throwing it as an `IllegalArgumentException` with the appropriate error message. This will simplify the number of catch blocks that a client of the `getInputsAndDisplay()` method would need to have to handle all possible types of exceptions that could arise from the method. A client of the `getInputsAndDisplay()` method would only need to catch `IllegalArgumentException`, as declared in the method header.

We have also included a finally block that lets the user know that the `getInputsAndDisplay()` method had finished running its full course, regardless of whether the program gracefully exited upon exception or successfully completed its function.

The client of `getInputsAndDisplay()` is the `main()` method, as shown in **Cell A1**:

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    try {
        getInputsAndDisplay(input);
    } catch (IllegalArgumentException e) {
        System.out.println(e.getMessage());
    } finally {
        System.out.println("Ready to do another task in main()");
    }
}
```

We invoke the `getInputsAndDisplay()` method. It could throw `IllegalArgumentException`, as declared in its method header, and we choose to handle the unchecked exception by printing the detailed error message. After this task in `main()` is finished executing, we let the user know that the program is ready for another task.

To see the chained exception, we provide a "bad" input of the zero divisor, as follows:

```
Enter two integers for division: A/B
A =
10
B =
0
Finished getInputsAndDisplay()
ArithmeticException: Divisor cannot be zero
Ready to do another task in main()
```

Let's break down what happened here.

First, the `getInputsAndDisplay()` invocation in main prompted the user for input. It saved the variables `a = 10` and `b = 0`. Then, `getInputsAndDisplay()` invoked the `doDivision()` method. Figure 7-4a depicts the flow of method invocations.

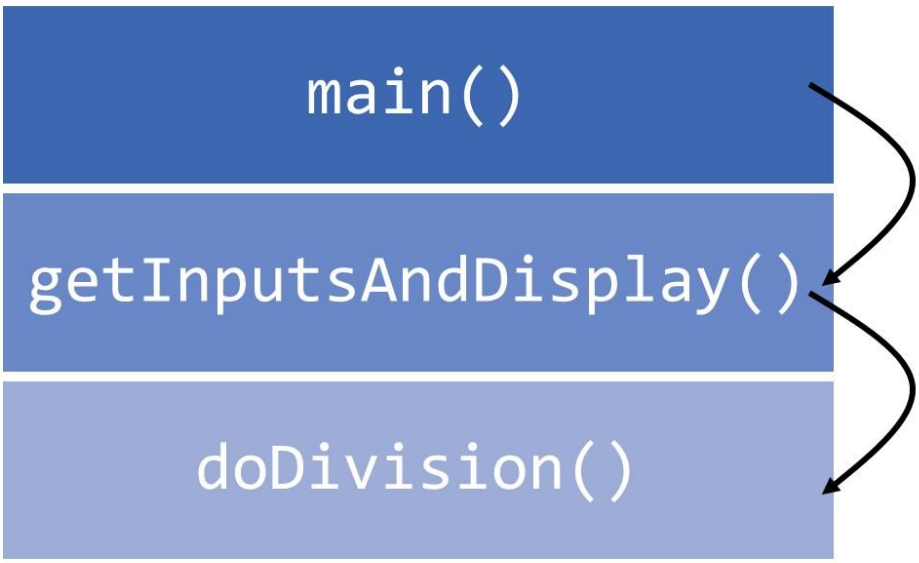

Figure 7-4a Flow of method invocations in ***JavaExceptionsEx3***

Upon the execution of `doDivision()`, an `ArithmeticException` is thrown due to division by zero. The `doDivision()` method had declared the `ArithmeticException`, so we know that it is not handled by the method. Instead, the exception propagates upward and the method that invoked `doDivision()` must handle it. Figure 7-4b illustrates what happens upon the `ArithmeticException`.

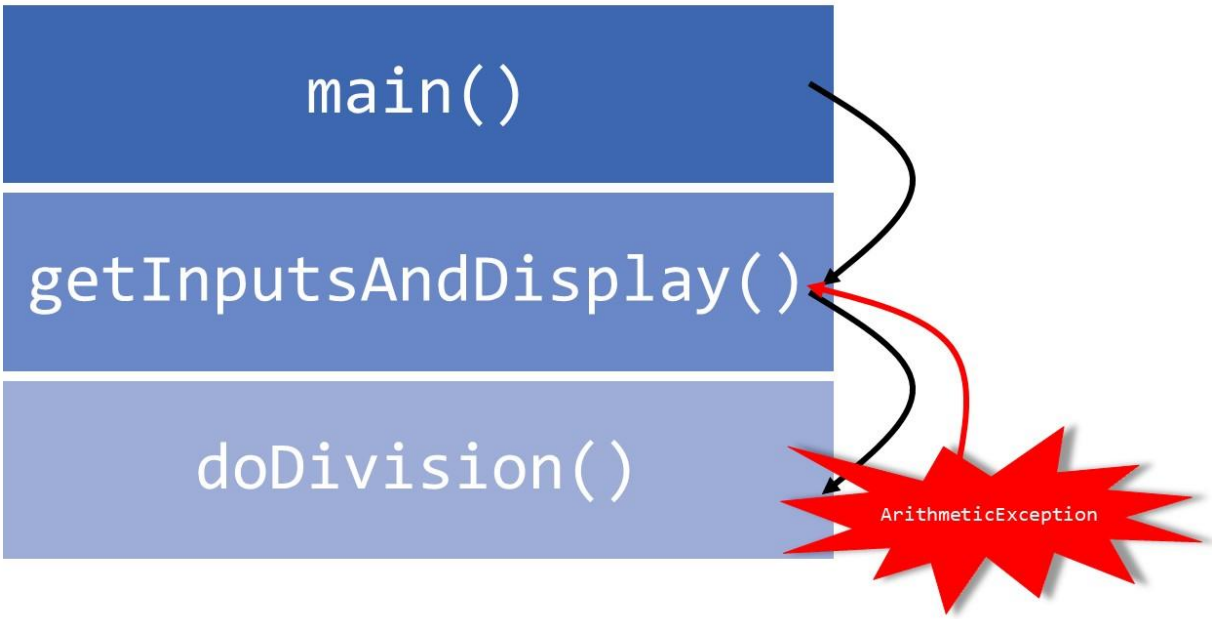

Figure 7-4b An `ArithmeticException` *is thrown by the* `doDivision()` *method.*

Fortunately, the method that "receives" the `ArithmeticException` has a catch block for it. The `getInputsAndDisplay()` method will catch the `ArithmeticException` and turn it into an `IllegalArgumentException`. The `IllegalArgumentException` is propagated upward to the client of `getInputsAndDisplay()`. This process is depicted in Figure 7-4c.

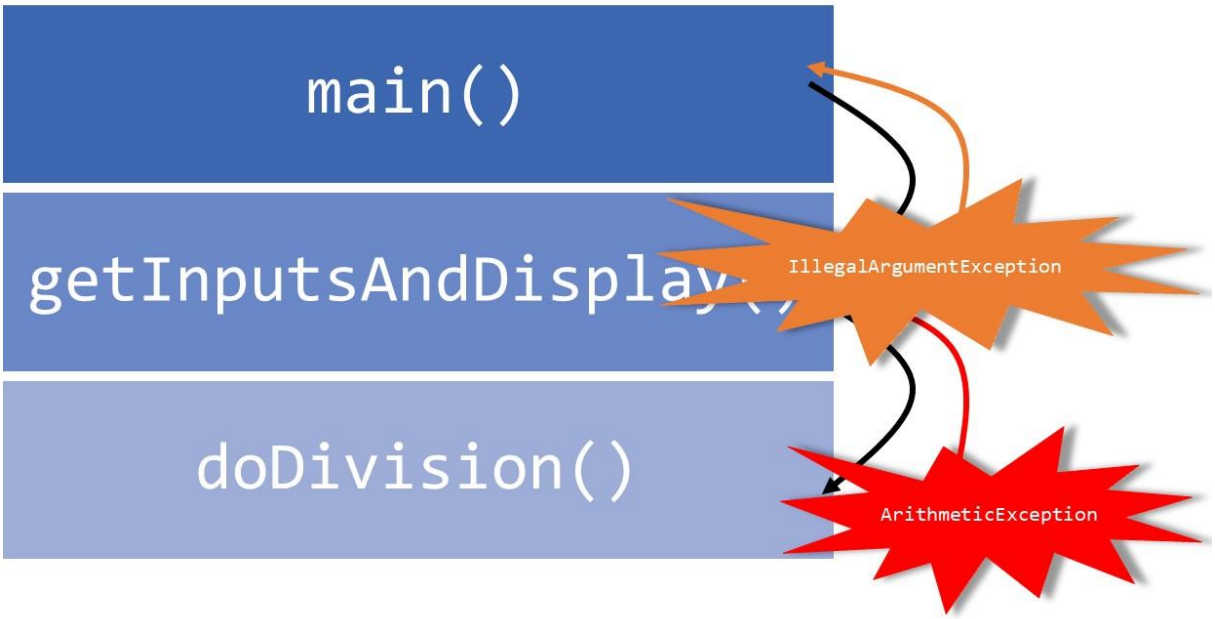

Figure 7-4c Chained exception through `getInputsAndDisplay()`

Because an `Exception` occurred during `getInputsAndDisplay()`, the method cannot finish executing. It skips the printing of the division result, but after the catch block executes, the program will continue to execute the finally block. Therefore, the finishing statement for `getInputsAndDisplay()` will print.

After the finally block for `getInputsAndDisplay()` finishes, the program returns to finish the rest of the `main()` method. The `main()` method will handle the `IllegalArgumentException` that it receives. According to the catch block inside `main()`, the program will print the error message within the `IllegalArgumentException`. After finishing the catch block, the program will proceed to the finally block of `main()`. The complete process of the chained exception is depicted in Figure 7-4d.

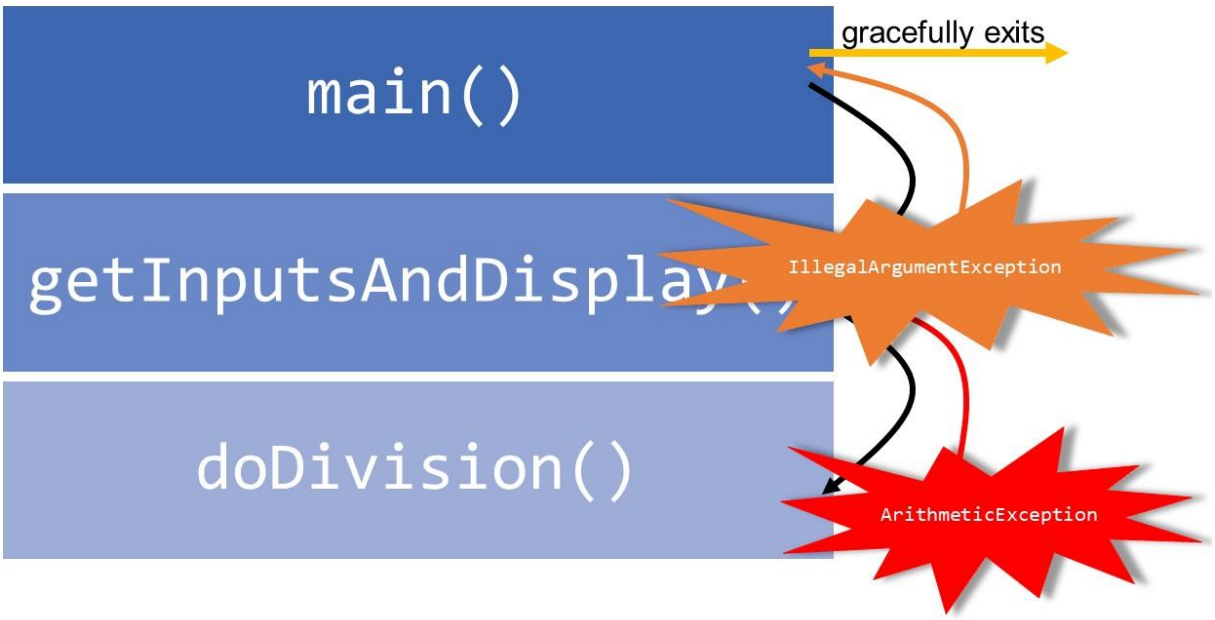

Figure 7-4d Program will gracefully exit upon `IllegalArgumentException` *in* `main()`

Now it's your turn! Try **Knowledge Check 4 for Exceptions:** Suppose that for the first line of the main() program, instead of

```
Scanner input = new Scanner(System.in);
```

We used a bad input:

```
Scanner input = null;
```

Then, we ran the program as previously. What should print? Don't run the program! Use logic and what you know about chained exceptions!

7.6 Creating Customized Exceptions

All the exceptions we have seen up to now are part of Java's libraries. They are not the only exceptions that a program may raise. **You can also define your own exception objects by**

extending one of the predefined classes in the Java exception class hierarchy. In this section, we will revisit **JavaExceptionEx2**, the file processing example that we completed earlier with `alice.txt`, and create a customized exception.

JavaExceptionEx4: JavaExceptionEx4.java

Goal: Create a customized exception

```
package client;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import resources.*;

public class JavaExceptionEx4 {
```

A0

```
    public static void main(String[] args) {
        try {
            getInputsAndDisplay();
        } catch (IllegalArgumentException e) {
            System.out.printf("IllegalArgumentException occurred: \n"
                + "Name: %s\n"
                + "Message: %s\n"
                + "Cause: %s\n",
                e.toString(), e.getMessage(), e.getCause());
        } finally {
            System.out.println("Ready for another task");
        }
    }
```

A1

```
    public static File readFile(String filename)
        throws FileNotFoundException {

        if (filename == null) {
            throw new IllegalArgumentException("Filename provided is
                                                null.");
        }
        File file = new File(filename);
        if (!file.exists()) {
            throw new FileNotFoundException("Filename provided cannot be
                                             found.");
        }
        return file;
    }
```

```
    public static int countOccurrences(File file, String tgt)
        throws FileNotFoundException, NotAWordException {

        if (file == null || tgt == null) {
            throw new IllegalArgumentException("File or target String
                provided is null.");
        }
        if (tgt.length() == 0) {
            throw new IllegalArgumentException("Target String provided is
                empty.");
        }
        Pattern p = Pattern.compile("^[a-zA-Z]+$");
        Matcher m = p.matcher(tgt);
        if (!m.find()) {
            throw new NotAWordException("Target String provided is not a
                word");
        }
        Scanner in = new Scanner(file);
        String line;
        String[] words;
        int count = 0;
        while (in.hasNextLine()) {
            line = in.nextLine().trim();
            line = line.replaceAll("[.?!'-,;:\"*]", "");
            words = line.split(" ");
            for (String word: words) {
                word = word.trim();
                if (word.equals(tgt)) {
                    count++;
                }
            }
        }
        in.close();
        return count;
    }
```

A2

```
    public static void getInputsAndDisplay()
        throws IllegalArgumentException {

        Scanner in = new Scanner(System.in);
        System.out.println("Please enter a filename to process:");
        String filename = in.nextLine().trim();
        System.out.println("Please enter a target word to look for:");
        String tgt = in.nextLine().trim();
        try {
            File file = readFile(filename);
            System.out.printf("Number of occurrences of %s: %d\n",
                              tgt, countOccurrences(file, tgt));
```

A3

```
        } catch (FileNotFoundException e) {
            throw new IllegalArgumentException(
                String.format("File not found: %s", filename), e);
        } catch (NotAWordException e) {
            throw new IllegalArgumentException(e.getMessage(), e);
        } finally {
            in.close();
            System.out.println("Finished getInputsAndDisplay().");
        }
    }
}
```

A4

JavaExceptionEx4: NotAWordException.java

Goal: Create a customized exception

```
package resources;
import java.io.IOException;

@SuppressWarnings("serial")
public class NotAWordException extends IOException {
    public NotAWordException() {
        super();
    }

    public NotAWordException(String s) {
        super(s);
    }

    public NotAWordException(String s, Throwable cause) {
        super(s, cause);
    }
```

B1

```
    @Override
    public String toString() {
        return "NotAWordException: Customized IOException when the input "
            + "is not a word";
    }
}
```

B2

First, we define a new type of `Exception` that is a subtype of `IOException`. We call it the `NotAWordException`, and programs should throw it if the user input is invalid because it is not a "word," consisting of only alphabet letters.

To create the new type of `Exception`, we create a new class in the `resources` package and extend `IOException`, as in **Cell B1**:

```
package resources;
import java.io.IOException;

@SuppressWarnings("serial")
public class NotAWordException extends IOException {
    public NotAWordException() {
        super();
}

    public NotAWordException(String s) {
        super(s);
    }

    public NotAWordException(String s, Throwable cause) {
        super(s, cause);
    }
```

Don't worry about what the `@SuppressWarnings("serial")` means. It is there to prevent some compiler warnings (about serializable objects) from showing up.

The code above lays out three different constructors for the `NotAWordException`. We can create a "default" version of the object, one with a specific error message, or one with a tailored error message and the `NotAWordException`'s cause. **The `cause` parameter could be useful to track where the original exception had come from when using chained exceptions.**

Then, in **Cell B2**, we override the `toString()` method:

```
@Override
public String toString() {
    return "NotAWordException: Customized IOException when the input "
        + "is not a word";
}
```

We will use the `NotAWordException` during file processing.

The client driver lives in the `client` package, and we need to first import some classes from the Java libraries and our own packages, as in **Cell A0**:

```
package client;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import resources.*;

public class JavaExceptionEx4 {
```

The `main()` method will be the final executor of our program, and it calls the methods below it, so we will walk through those "helper" methods first.

The `readFile()` method did not change from before, when we worked through **JavaExceptionsEx2**.

We added some code to the `countOccurrences()` method, which is highlighted in yellow. The new lines in **Cell A2** are copied below for reference:

```
Pattern p = Pattern.compile("^[a-zA-Z]+$");
Matcher m = p.matcher(tgt);
if (!m.find()) {
    throw new NotAWordException("Target String provided is not a word");
}
```

We will count as a "word" only those `tgt` inputs that contain at least one alphabet character and no other types of characters. The easiest way to impose this restriction on the definition of "word" is to use a regular expression. We use the begin- and end-line anchors to say that the *entire* String must contain only alphabet letters. Then, we create the `Matcher` object and attempt to match the regular expression against the `tgt` input. If no matches are found, `tgt` was not a word, so we throw a `NotAWordException`.

Because `NotAWordException` is a subtype of `IOException`, it is a *checked* exception. We do not handle the `NotAWordException` within the `countOccurrences()` method, so we need to add it to the declaration in the method header:

```
public static int countOccurrences(File file, String tgt)
    throws FileNotFoundException, NotAWordException {
```

The next method, `getInputsAndDisplay()`, performs most of the functions of the original `main()` method from **JavaExceptionsEx2**. The `getInputsAndDisplay()` method uses the `readFile()` and `countOccurrences()` methods. See **Cell A3**:

```
public static void getInputsAndDisplay() throws IllegalArgumentException {
    Scanner in = new Scanner(System.in);
    System.out.println("Please enter a filename to process:");
    String filename = in.nextLine().trim();
    System.out.println("Please enter a target word to look for:");
    String tgt = in.nextLine().trim();
    try {
        File file = readFile(filename);
        System.out.printf("Number of occurrences of %s: %d\n",
                          tgt, countOccurrences(file, tgt));
```

In this method, we prompt the user for a filename and a target word, saving their inputs to String variables. Then, we attempt to create the `File` object by invoking `readFile()`, which may throw a `FileNotFoundException`, as declared in its method header. If the `File` object can be successfully created, we attempt to count the number of times the target word appears through the `countOccurrences()` method, which may throw a `FileNotFoundException` or a `NotAWordException`, as declared in its method header.

Should one of those checked exceptions (`FileNotFoundException` or `NotAWordException`) occur during the program execution, we provide catch blocks and apply the principle of chained exceptions to re-throw them outward. **Cell A4** does this:

```
    } catch (FileNotFoundException e) {
        throw new IllegalArgumentException(
            String.format("File not found: %s", filename), e);
    } catch (NotAWordException e) {
        throw new IllegalArgumentException(
            e.getMessage(), e);
    } finally {
        in.close();
        System.out.println("Finished getInputsAndDisplay().");
    }
}
```

If one of the two checked exceptions occur during a method call within `getInputsAndDisplay()`, we catch them and transform them into an `IllegalArgumentException`, which is an unchecked `RuntimeException`. Because the only `Exception` that could occur within `getInputsAndDisplay()` is now the `IllegalArgumentException`, we do not have to handle the unchecked exception. We chose, however, to declare the `IllegalArgumentException` in the method header to make it clear to clients of `getInputsAndDisplay()` what type of `Exception` they should be aware of.

Within the catch blocks, you'll see that we create new `IllegalArgumentException` objects with a detailed error message and a second parameter. We used the constructor with the `Throwable cause` parameter to track what had caused *this* `IllegalArgumentException`. We said before that the additional `cause` parameter may be useful to track chained exceptions, and this code shows an example. We directly feed in the thrown `Exception`, `e`, to the `cause` parameter, so that clients of `getInputsAndDisplay()` can see a "history" of where the `IllegalArgumentException` had come from.

As before, we also provide a finally block that lets the user know that the `getInputsAndDisplay()` method has finished running.

The "top-level" client method of our program will be the `main()` method, which calls `getInputsAndDisplay()`, as in **Cell A1**:

```
public static void main(String[] args) {
    try {
        getInputsAndDisplay();
    } catch (IllegalArgumentException e) {
        System.out.printf("IllegalArgumentException occurred: \n"
                + "Name: %s\n"
                + "Message: %s\n"
                + "Cause: %s\n",
                e.toString(), e.getMessage(), e.getCause());
    } finally {
        System.out.println("Ready for another task");
    }

}
```

The `main()` program tries to execute the operations of `getInputsAndDisplay()`. If it runs into the method's declared `IllegalArgumentException`, it prints out a very detailed report consisting of the `IllegalArgumentException`'s String representation, its error message, and its `cause`'s String representation. By default, the String representation of an `Exception` will be the name of the `Exception`'s class, followed by a colon and the `Exception`'s String representation.

We are now ready to run the program!

If we feed in a target word that consists of non-alphabet characters, a `NotAWordException` will get thrown by the `countOccurrences()` method, illustrated in Figure 7-5a.

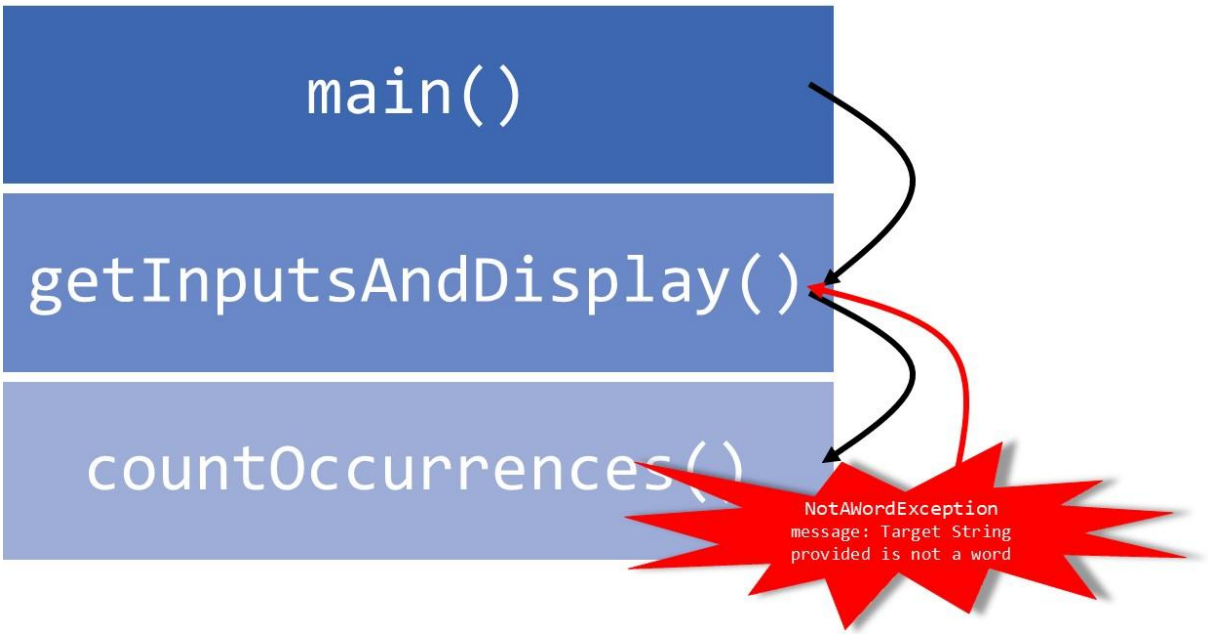

Figure 7-5a `NotAWordException` *occurs if target word has non-alphabet characters*

The `NotAWordException` will be caught by the `getInputsAndDisplay()` method and thrown upward as an `IllegalArgumentException`, keeping the original `NotAWordException` tagged as the cause. The `main()` program will catch the `IllegalArgumentException` from `getInputsAndDisplay()` and print out the detailed error report. This process is depicted in Figure 7-5b.

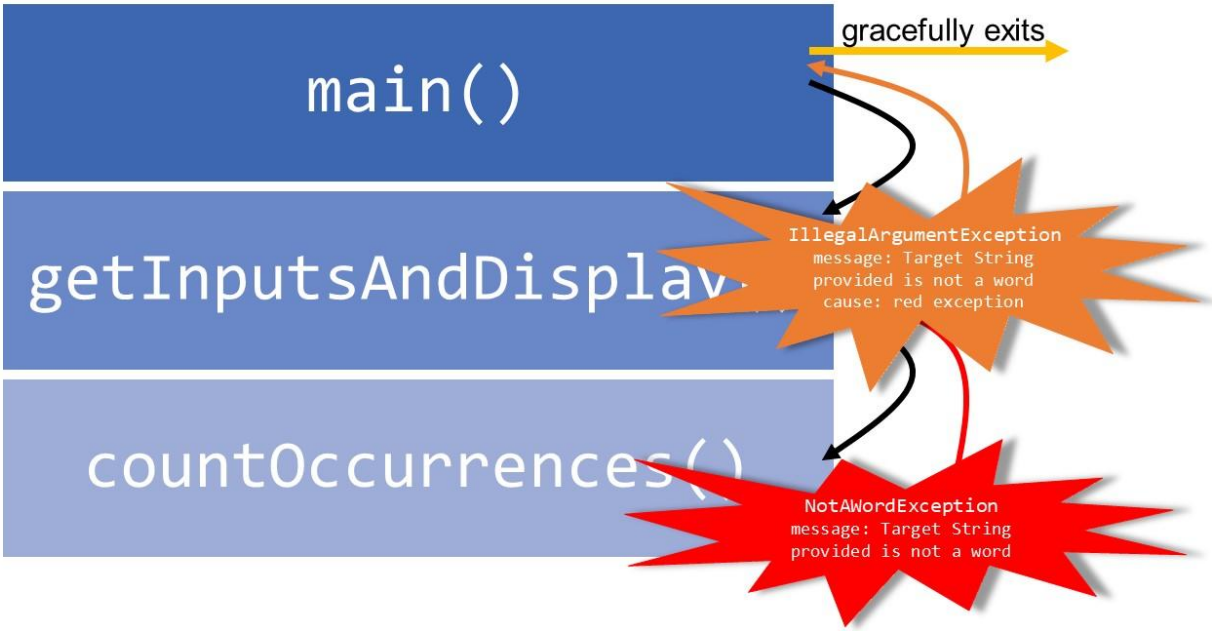

Figure 7-5b Clients handle the original `NotAWordException` *and chained exceptions*

The output of the program, then, with a non-word target is:

```
Please enter a filename to process:
files/alice.txt
Please enter a target word to look for:
Alice0
Finished getInputsAndDisplay().
IllegalArgumentException occurred:
Name: java.lang.IllegalArgumentException: Target String provided is not a word
Message: Target String provided is not a word
Cause: NotAWordException: Customized IOException when the input is not a word
Ready for another task
```

Now it's your turn! Try **Knowledge Check 5 for Exceptions:** Suppose that we provide the following inputs to the program:

```
Please enter a filename to process:
alice.txt
Please enter a target word to look for:
Alice0
```

Then, we ran the program as previously. What should print? Don't run the program! Use logic and what you know about chained exceptions and customized exceptions!

This concludes our discussion on Java exceptions! We will be using the principles of exception handling in the next chapter, when we deal with input/output, or IO.

7.7 More on Java Secure Coding Guidelines

In SEI CERT Oracle Coding Standard for Java, its rule 07--Exceptional Behavior (ERR) is about secured coding on exception.
(https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88487665)
For ERR00-J, "Do not suppress or ignore checked exceptions" recommends that for checked exception, don't leave the catch block empty or do trivial task. Vice versa, make sure the catch block does some further work to recover from the exceptional condition, rethrow the exception to allow the next nearest enclosing catch clause of a try statement to recover, or throw an exception that is appropriate to the context of the catch block.

In our JavaException_SecuredEx example, we ask user to enter fruits.txt file name and then retrieve the first number 99 from the file:

```
public static void main(String[] args) {
    String fruitFile;
```

```
Scanner input = new Scanner(System.in);
try{
    System.out.println("please enter a correct file name:");
    fruitFile = input.next();//"fruits.txt";
    File f = new File(fruitFile);
    Scanner contents = new Scanner(f);
    contents.nextLine();//skip the first line
    String[] words = contents.nextLine().split(" ");
    contents.close();
    int aFruit = Integer.parseInt(words[3]);
    System.out.println("the number of fruit is: " + aFruit);
}
catch(FileNotFoundException ex){
    System.out.println(ex);
    ex.printStackTrace();
}
```

Since the program needs to deal with opening file and FileNotFoundException is a checked exception, the catch block outputs the exception. When we run the program and feed a wrong file name:

```
please enter a correct file name:
C:\Users\ziping\Documents\fruits.txt
java.io.FileNotFoundException: C:\Users\ziping\Documents\fruits.txt (The
system cannot find the file specified)
java.io.FileNotFoundException: C:\Users\ziping\Documents\fruits.txt (The
system cannot find the file specified)
        at java.io.FileInputStream.open0(Native Method)
        at java.io.FileInputStream.open(Unknown Source)
        at java.io.FileInputStream.<init>(Unknown Source)
        at java.util.Scanner.<init>(Unknown Source)
        at
javaexception_securedex.JavaException_SecuredEx.main(JavaException_SecuredEx.j
ava:29)
```

the program output the exception and exited. The above design is to ignore the exception and will not give users a second chance to try with a correct file name. In order to give a user a couple of more tries, we revised the program as:

```
boolean validFile = false;
for(int i = 0; i < 3; i++){
    try{
        System.out.println("please enter a correct file name:");
        fruitFile = input.next();//"fruits.txt";
        File f = new File(fruitFile);
        Scanner contents = new Scanner(f);
        contents.nextLine();//skip the first line
        String[] words = contents.nextLine().split(" ");
        contents.close();
        int aFruit = Integer.parseInt(words[3]);
        System.out.println("the number of fruit is: " + aFruit);
        validFile = true;
        if(validFile)
```

```
                break;
            }
            catch(FileNotFoundException ex){
                System.out.println("wrong file name!");
                if(i == 2)
                    System.out.println("Three wrong entries. Program ends.");
            }
        }
```

And when we run the program, we have the following running result:

```
please enter a correct file name:
f1
wrong file name!
please enter a correct file name:
f2
wrong file name!
please enter a correct file name:
fruits.txt
the number of fruit is: 99
```

Our revised version can give users more chances for a correct file name input and when a user fails in three trials, the program will exit.

In rule ERR01-J, “Do not allow exceptions to expose sensitive information”, it is recommended that programs sanitize exceptions containing sensitive information. For instance, FileNotFoundException can give out file related information such as specified file name and file path as well as its backtrace to a specified output stream, of which attackers can take advantage to collect those sensitive information by trying various file names

In our previous discussed JavaException_SecuredEx example, we throw FileNotFoundException and the catch block outputs the exception as well as its backtrace:

```
        catch(FileNotFoundException ex){
            System.out.println(ex);
            ex.printStackTrace();
        }
```

From the running result we can see that the exception, the Java file name and the line of Java code are all revealed when the wrong file location is provided:

```
please enter a correct file name:
C:\Users\ziping\Documents\fruits.txt
java.io.FileNotFoundException: C:\Users\ziping\Documents\fruits.txt (The
system cannot find the file specified)
java.io.FileNotFoundException: C:\Users\ziping\Documents\fruits.txt (The
system cannot find the file specified)
        at java.io.FileInputStream.open0(Native Method)
        at java.io.FileInputStream.open(Unknown Source)
        at java.io.FileInputStream.<init>(Unknown Source)
        at java.util.Scanner.<init>(Unknown Source)
```

```
        at
javaexception_securedex.JavaException_SecuredEx.main(JavaException_SecuredEx.j
ava:29)
```

In rule ERR04-J, "Do not complete abruptly from a finally block" (https://wiki.sei.cmu.edu/confluence/display/java/ERR04-J.+Do+not+complete+abruptly+from+a+finally+block), it is recommended not use return, break, continue, or throw statements within a finally block to prevent the suppressing of an exception.

In our JavaException_SecuredEx example, doDivisionAndDisplay( ) method's finally block has a return statement:

```
    public static void doDivisionAndDisplay(Scanner input) {
        try {
            System.out.println("Enter two integers for division: A/B");
            System.out.println("A = ");
            int a = input.nextInt();
            System.out.println("B = ");
            int b = input.nextInt();
            System.out.println(a + " / " + b + " is " + (a / b));
        } catch (ArithmeticException e) {
            System.out.println("Catch ArithmeticException: "
                    + "Divisor cannot be zero ");
        } finally {
            System.out.println("Ready to do another task");
            return;
        }

        //return; //place the return statement here instead
    }
```

The running result is given as below:

```
Enter two integers for division: A/B
A =
q
Ready to do another task
```

We entered a letter instead of an integer number and the method terminates abruptly. If we take away the finally block, the running result is:

```
Enter two integers for division: A/B
A =
q
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at
javaexception_securedex.JavaException_SecuredEx.doDivisionAndDisplay(JavaExcep
tion_SecuredEx.java:72)
```

```
        at
javaexception_securedex.JavaException_SecuredEx.main(JavaException_SecuredEx.j
ava:65)
```

We can see that there are other exceptions that are not included for consideration, so the design is not ideal.