

Advanced Java, part 2

CS121: Data Structures

START RECORDING

Attendance Quiz

Attendance Quiz: Inheritance and Interfaces

- Scan the QR code, or find today's attendance quiz under the "Quizzes" tab on Canvas
- Password: to be announced



Attendance Quiz: Inheritance and Interfaces

- Write your name and the date
- Briefly describe two differences between inheritance and interfaces in Java
- Create two versions of a Laptop class (no need to define any methods):
 - One version should **inherit** from a Computer **class**
 - One version should **implement** a Computer **interface**

Outline

- Attendance quiz
- Exceptions
- User-friendly error messages

Exception Handling in Java

Slides based on those from Richard Huntrods, P.Eng.

Exception Handling in Java

- Topics:
 - Introduction
 - Historic approaches to error handling
 - Modern error handling with exceptions
 - Types of exceptions
 - Coding exceptions

Introduction

- Users (and other programmers) will use our programs in unexpected ways
- Users want our programs to behave predictably (e.g., a word processor shouldn't lose your edits)
- But due to unexpected situations, design errors, or coding errors, our programs may fail in unexpected ways during execution

Introduction

- We should write quality code that does not fail catastrophically. It is bad for programs to crash:
 - Without interpretable error messages
 - Leaving an operation partially completed
- Even worse is when a program continues running, but produces incorrect results, corrupts files, etc.
- Consequently, we must design error handling into our programs

Division Example

- How many ways can you crash this program?
 - Not enough arguments (ArrayIndexOutOfBoundsException)
 - Not integers (NumberFormatException)
 - Division by zero (ArithmeticException)
- Exception handling gives us control over what happens when these problems occur

```
public class Division {  
    public static void main(String[] args) {  
        System.out.println(Integer.parseInt(args[0]) / Integer.parseInt(args[1]));  
    }  
}
```

Errors and Error Handling

- An Error is any unexpected result obtained from a program during execution
- Unhandled errors may manifest themselves as incorrect results or behavior, or as abnormal program termination
- Ideally, errors should be handled by the programmer, to prevent them from reaching the user

Errors and Error Handling

- Some typical causes of errors:
 - Calculation errors (i.e. divide by 0)
 - Array errors (i.e. accessing element -1)
 - Conversion errors (i.e. convert 'q' to a number)
 - Memory errors (i.e. memory incorrectly allocated, memory leaks, "null pointer" – common in C, C++, etc.)
 - File system errors (i.e. disk is full, disk has been removed)
 - Network errors (i.e. network is down, URL does not exist)
 - Can you think of some others?

Traditional Error Handling: Error Flags

- Every method returns a value (flag) indicating either success, failure, or some error condition. The calling method checks the return flag and takes appropriate action.
- For example: C uses this method for almost all library functions (i.e. `fopen()` returns a valid file or else null)
- **Downside:** programmer must remember to always check the return value and take appropriate action. This requires adding conditionals (making methods harder to read), and it's easy to forget to perform these checks.

Traditional Error Handling: Global Error Handler

- Create a global error handling routine, and use some form of “jump” instruction to call this routine when an error occurs.
- For example: many older programming texts (for C, FORTRAN) recommended this method. Those who use this method will frequently adapt it to new languages (C++, Java).
- **Downside:** “jump” instruction (GoTo) is considered a “bad programming practice” and is discouraged. Once you jump to the error routine, it is hard to return to the point of origin and so you will (probably) exit the program. “jump” leads to unmaintainable code, because program flow is unpredictable.

Modern Error Handling: Exceptions

- Exceptions are a mechanism that provides the best of both worlds
- Like method return flags, any method may raise an exception if it encounters an error, and your code can decide how to proceed, but:
 - Handling of certain exceptions can be enforced
 - Exception handling is clearly differentiated from other code
- Like jumping to a global error handler, exceptions allow you to skip code when an error is encountered
 - But exceptions don't obfuscate program flow

Exceptions in Java

- An exception is a representation of an error condition or a situation that is not the expected result of a method.
- Exceptions are built into the Java language and are available to all program code.
- Exceptions isolate the code that deals with the error condition from regular program logic.

Types of Exceptions

- In Java, exceptions fall into two categories:
 - Unchecked Exceptions (which we saw in the Division example)
 - Checked Exceptions

Unchecked Exceptions

- Inherit from `java.lang.RuntimeException`
- Intended to represent error conditions that are considered “fatal” to program execution
- You do not have to do anything with an unchecked exception
 - By default, your program will terminate (crash) with an appropriate error message
 - Though you can choose to handle them

Checked Exceptions

- Inherit from `java.lang.Exception`
- Intended to represent exceptions that are usually “non fatal” to program execution
- Checked exceptions **must** be handled in your code, or explicitly passed to parent classes for handling

Examples

Unchecked exceptions:

- `ArrayIndexOutOfBoundsException`
- `NumberFormatException`
- `ArithmeticException`
- ...

Checked exceptions:

- `java.io.FileNotFoundException`
- `java.io.IOException`
- `java.net.http.HttpTimeoutException`
- ...

Handling Exceptions

- Exception handling is accomplished through:
 - The “try – catch” mechanism, or
 - A “throws” clause in the method declaration
- If you call any methods that throw a checked exception, you must decide whether to handle the exception yourself, or pass the exception “up the chain” to the calling method

Try-Catch(-Finally)

- try: the code that might trigger an exception is placed inside a block of code starting with the “try” keyword
- catch: the code to handle the exception (should it arise) is placed in a block of code starting with the “catch” keyword
- finally: a “finally” block of code can be optionally include, which will ALWAYS be executed, either after the “try” block code, or after the “catch” block code
- Useful for operations that must happen no matter what (i.e. cleanup operations such as closing a file)

Try-Catch-Finally Example

```
try {  
    // normal program code  
}  
catch(ArithmeticException e) {  
    // exception handling code  
}  
finally {  
    // cleanup code  
}
```


Division with Exception Handling

```
public class PoliteDivision {
    public static void main(String[] args) {
        try {
            System.out.println(Integer.parseInt(args[0]) / Integer.parseInt(args[1]));
        }
        catch (ArrayIndexOutOfBoundsException | NumberFormatException e) {
            System.out.println("Please supply two integers.");
        }
        catch (ArithmeticException e) {
            System.out.println("Division by zero is undefined");
        }
    }
}
```

Passing an Exception

- In any method that might throw an exception, you may declare that the method “throws” that exception, and thus avoid handling the exception yourself
- Example
 - ```
public void myMethod throws IOException {
 // normal code with some I/O
}
```

# Exception Classes

- All checked exceptions have class “Exception” as the parent class.
- You can use the actual exception class or the parent class when referring to an exception
- Where do you find the exception classes?
  - [Java's documentation for the Exception class](#)

# Reading a File

- The built-in Java classes for reading files include checked exceptions
- This means that all code must either handle the exception, or throw it
- Probably motivated the textbook authors to create the In class!

# Reading a File

- Catching the exception allows us to print an intelligible error message

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Cat {
 public static void main(String[] args) {
 try {
 Scanner s = new Scanner(new File(args[0]), "UTF-8");
 while (s.hasNextLine()) {
 System.out.println(s.nextLine());
 }
 s.close();
 }
 catch (FileNotFoundException e) {
 System.out.println("Could not read file: " + args[0]);
 }
 }
}
```

- Refactored into a method

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class CatMethod {

 public static void cat(String filename) throws FileNotFoundException {
 Scanner s = new Scanner(new File(filename), "UTF-8");
 while (s.hasNextLine()) {
 System.out.println(s.nextLine());
 }
 s.close();
 }

 public static void main(String[] args) {
 try {
 cat(args[0]);
 }
 catch (FileNotFoundException e) {
 System.out.println("Could not read file: " + args[0]);
 }
 }
}
```

- Simply throwing the exception
- This will cause the program to crash, with the default stack trace

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class CatThrows {
 public static void main(String[] args) throws FileNotFoundException {
 Scanner s = new Scanner(new File(args[0]), "UTF-8");
 while (s.hasNextLine()) {
 System.out.println(s.nextLine());
 }
 s.close();
 }
}
```

- Throwing all exceptions
- **A bad practice,** because it makes it unclear which errors the program can encounter

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class CatThrows {
 public static void main(String[] args) throws Exception {
 Scanner s = new Scanner(new File(args[0]), "UTF-8");
 while (s.hasNextLine()) {
 System.out.println(s.nextLine());
 }
 s.close();
 }
}
```



```
import java.io.File;
import java.util.Scanner;
```

```
public class CatBad {
 public static void main(String[] args) {
 Scanner s = new Scanner(new File(args[0]), "UTF-8");
 }
}
```

Unhandled exception: java.io.FileNotFoundException

[Add exception to method signature](#)   [More actions...](#)  

**java.util.Scanner**  
public Scanner([File](#) source,  
                  [String](#) charsetName)  
                  throws [FileNotFoundException](#)

Constructs a new Scanner that produces values scanned from the specified file. Bytes from the file are converted into characters using the specified charset.

Parameters:


source - A file to be scanned

charsetName - The encoding type used to convert bytes from the file into characters to be scanned

Throws:

[FileNotFoundException](#) - if source is not found

[IllegalArgumentException](#) - if the specified encoding is not found

 < 11 >

[External documentation for `Scanner\(File, String\)`](#) 

# DO NOT Catch All Exceptions

- Catching all exceptions
- **A terrible practice**, because we may catch exceptions we don't want to (e.g., the user pressing Control-C to cleanly exit the program, etc.), allowing program execution to continue after any kind of error

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Cat {
 public static void main(String[] args) {
 try {
 Scanner s = new Scanner(new File(args[0]), "UTF-8");
 while (s.hasNextLine()) {
 System.out.println(s.nextLine());
 }
 s.close();
 }
 catch (Exception e) {
 System.out.println("Could not find file: " + args[0]);
 }
 }
}
```

# Where Do Exceptions Come From?

- When a method encounters an error (e.g., trying to open a non-existent file), an exception is raised
- But where does that exception come from?
  - Code can use the `throw` keyword to propagate an exception

```
public class President {
 private static final String[] presidents = {
 "George Washington",
 ...
 "Joseph R. Biden Jr."
 };

 public final String name;

 public President(int n) {
 if (n <= presidents.length && n > 0) name = presidents[n - 1];
 else throw new IllegalArgumentException("n is not between 1 and " + presidents.length);
 }

 public static void main(String[] args) {
 while (true) {
 StdOut.printf("Which president (1-%d)? ", presidents.length);
 try {
 President p = new President(StdIn.readInt());
 StdOut.println("You selected: " + p.name);
 }
 catch (IllegalArgumentException e) {
 StdOut.println("Please type a number between 1 and " + presidents.length);
 }
 }
 }
}
```

```
public President(int n) {
 if (n <= presidents.length && n > 0) name = presidents[n - 1];
 else throw new IllegalArgumentException("n is not between 1 and " + presidents.length);
}
```

```
public President(int n) {
 try {
 name = presidents[n - 1];
 }
 catch (ArrayIndexOutOfBoundsException e) {
 throw new IllegalArgumentException("n is not between 1 and " + presidents.length, e);
 }
}
```

**Equivalent,  
but less likely  
to have an off-  
by-one error**

- Why catch an exception, only to throw a different exception?
  - Good to hide implementation details
  - Users shouldn't need to know that we're using an array: it's better to throw `IllegalArgumentException` instead of an `ArrayIndexOutOfBoundsException`

# Summary

- Exceptions are a powerful error handling mechanism built in to Java
- Exceptions can be handled locally (try-catch), or handled by the calling code (throws)

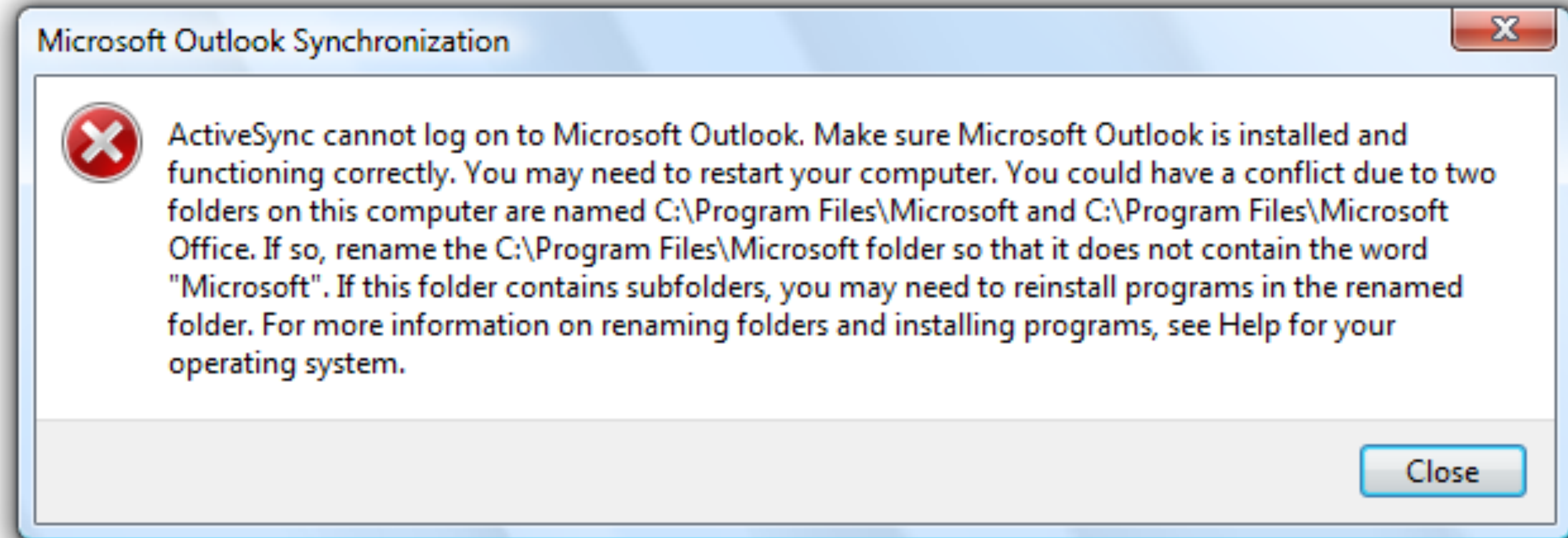
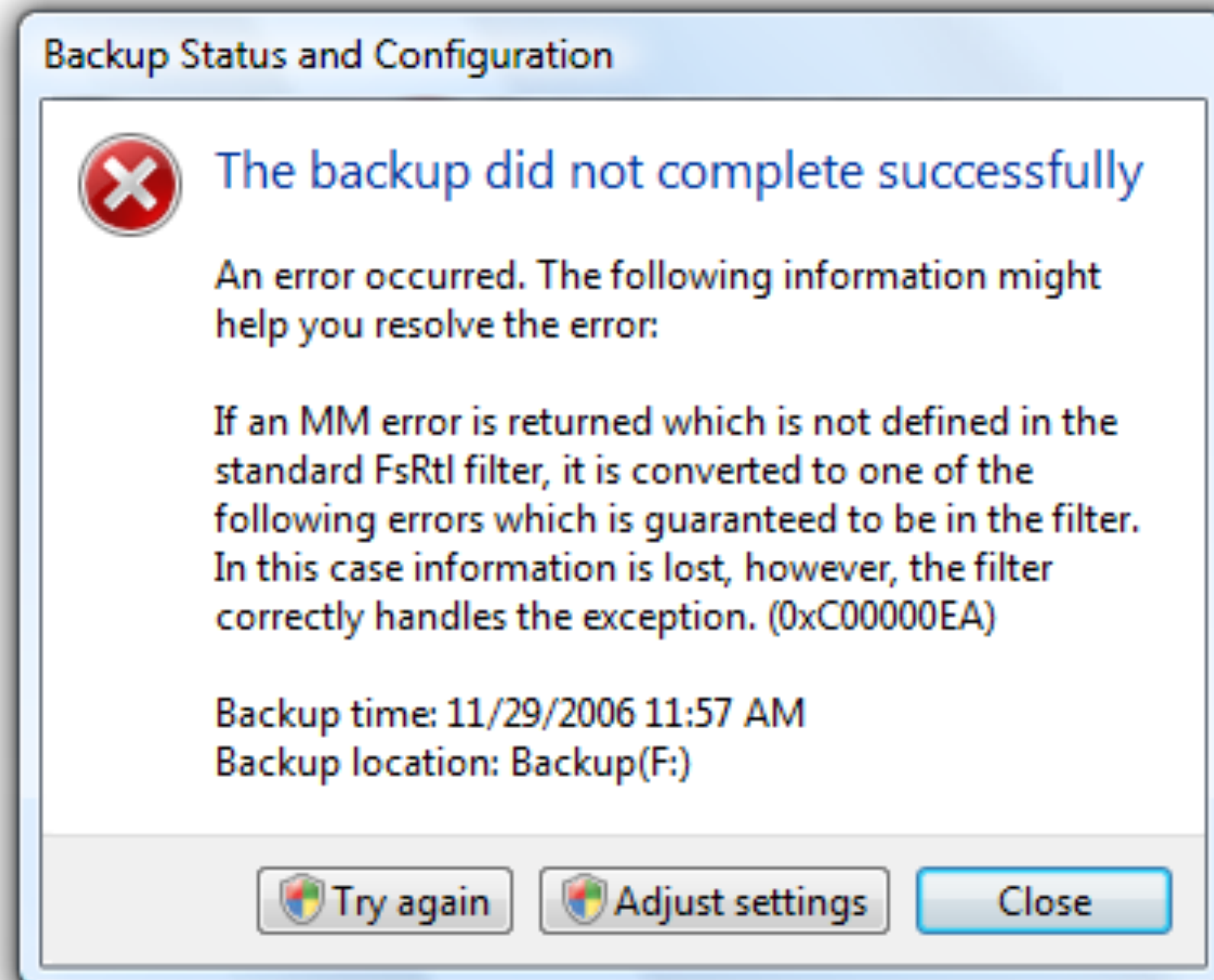
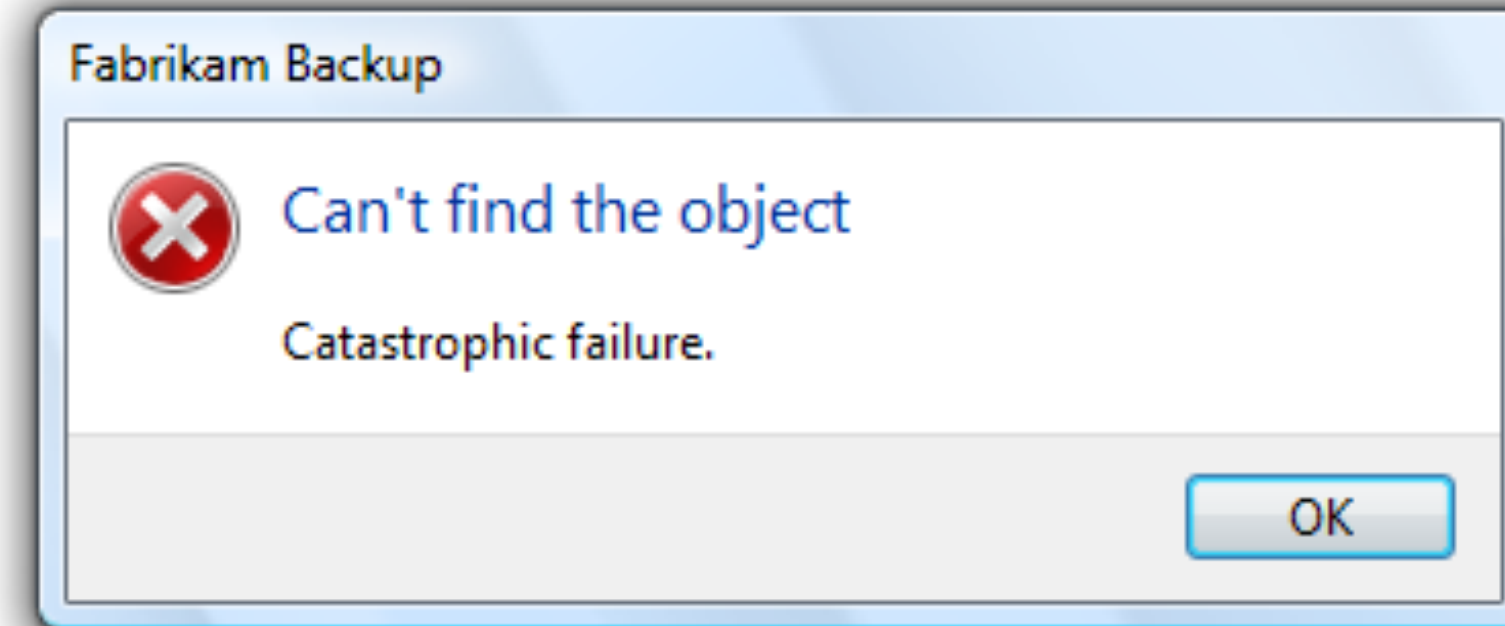
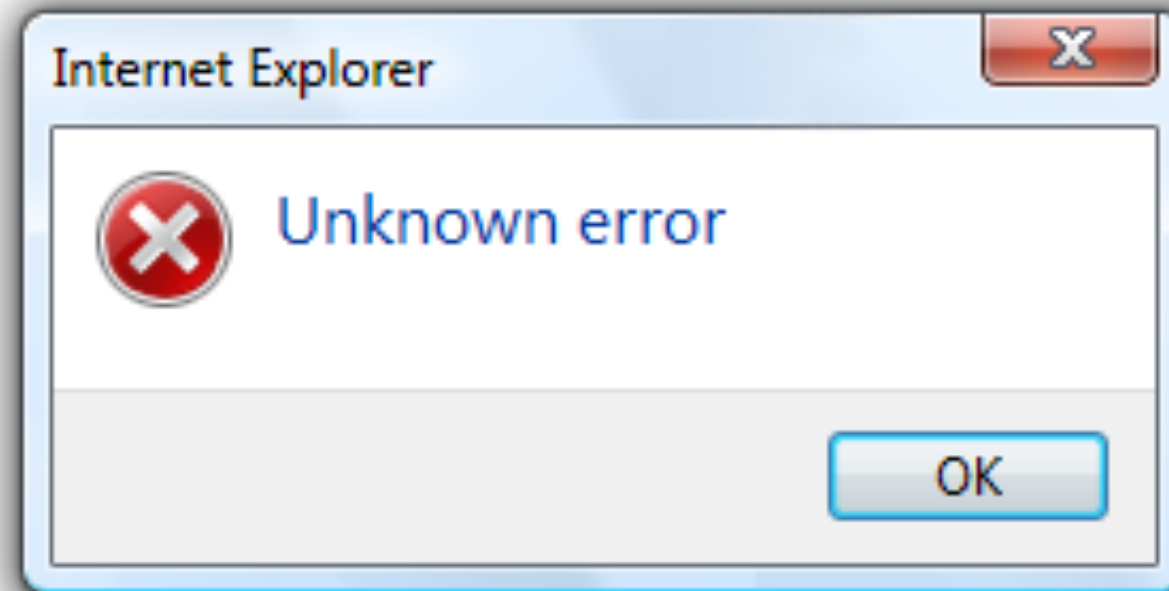
# Further Reading

- Joshua Bloch's "Effective Java, 3rd Edition" (includes a whole chapter on exceptions)

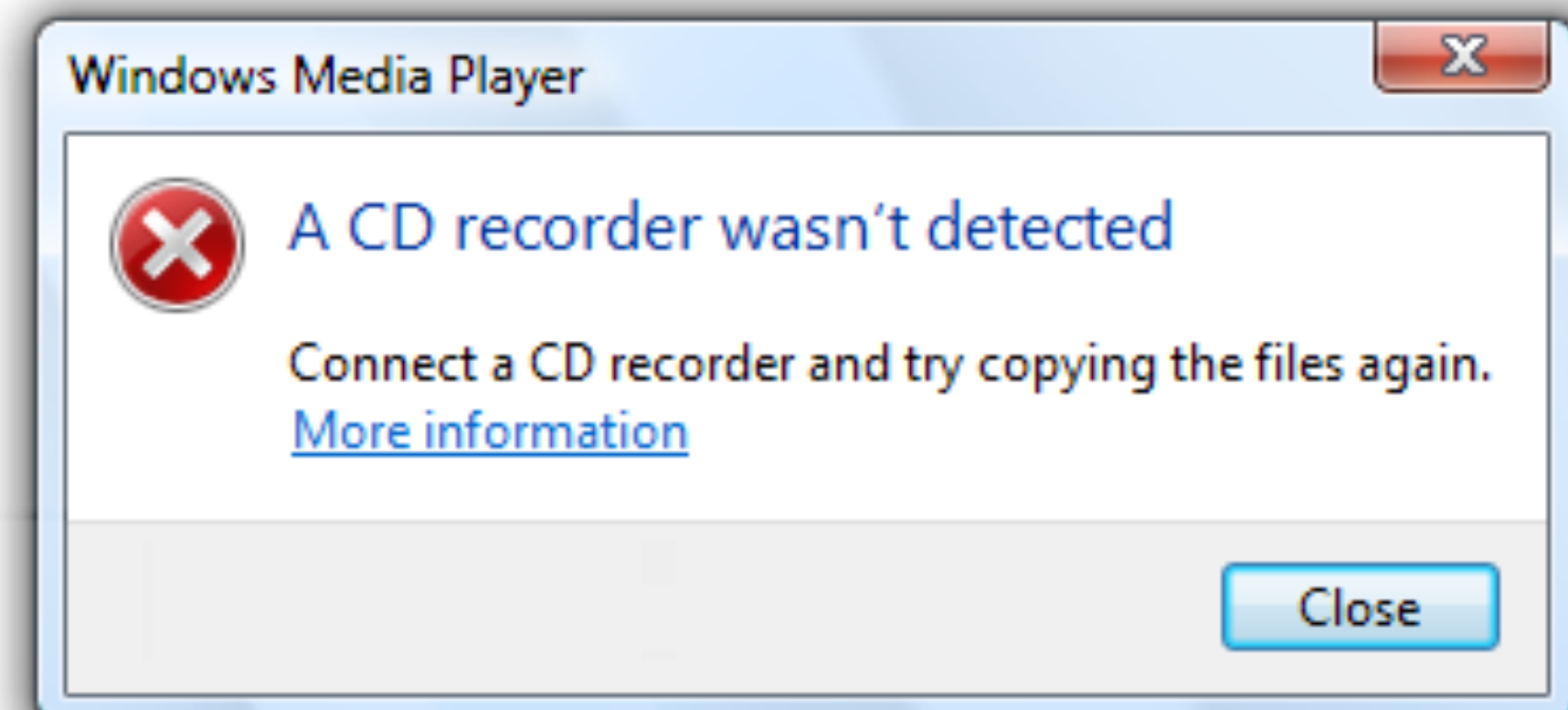
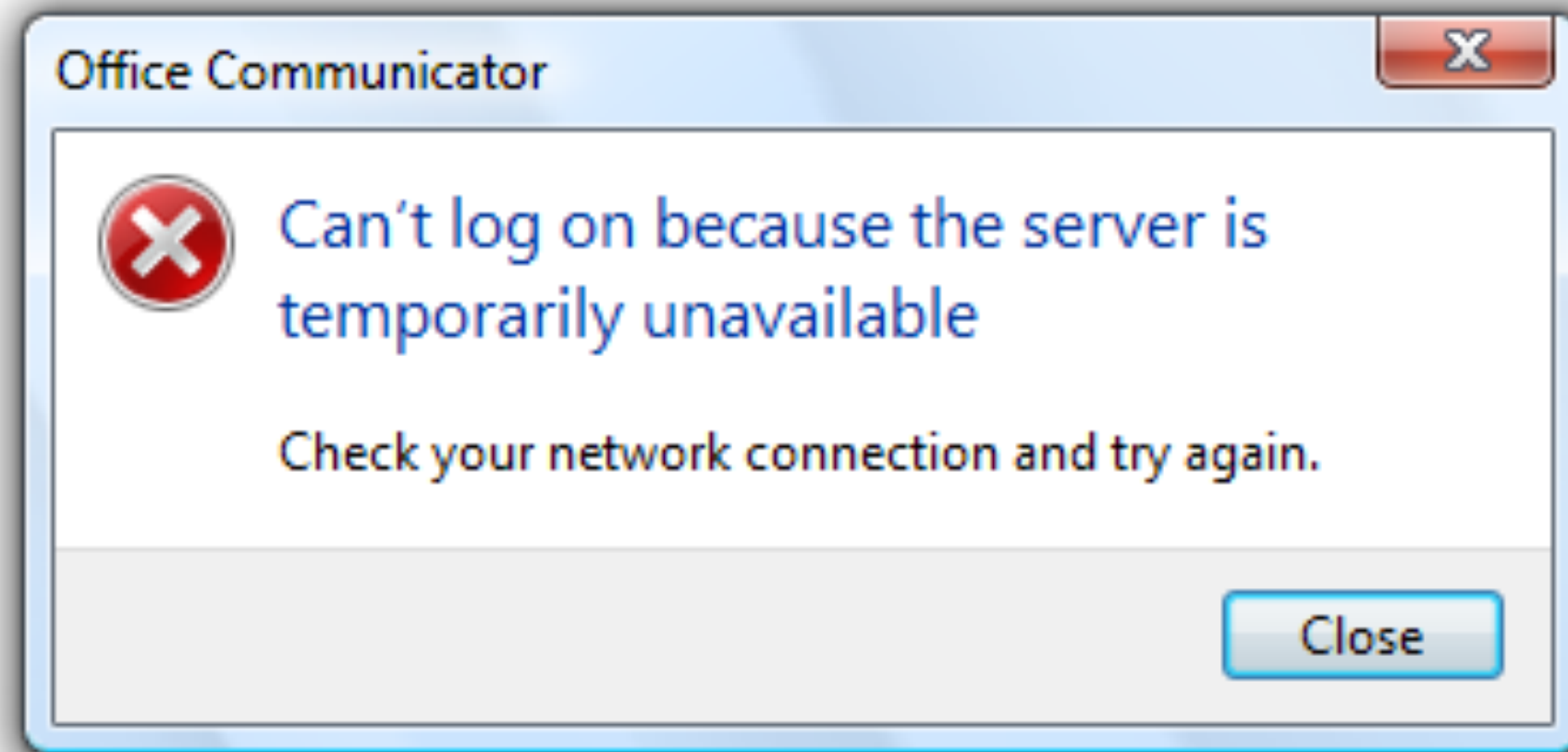
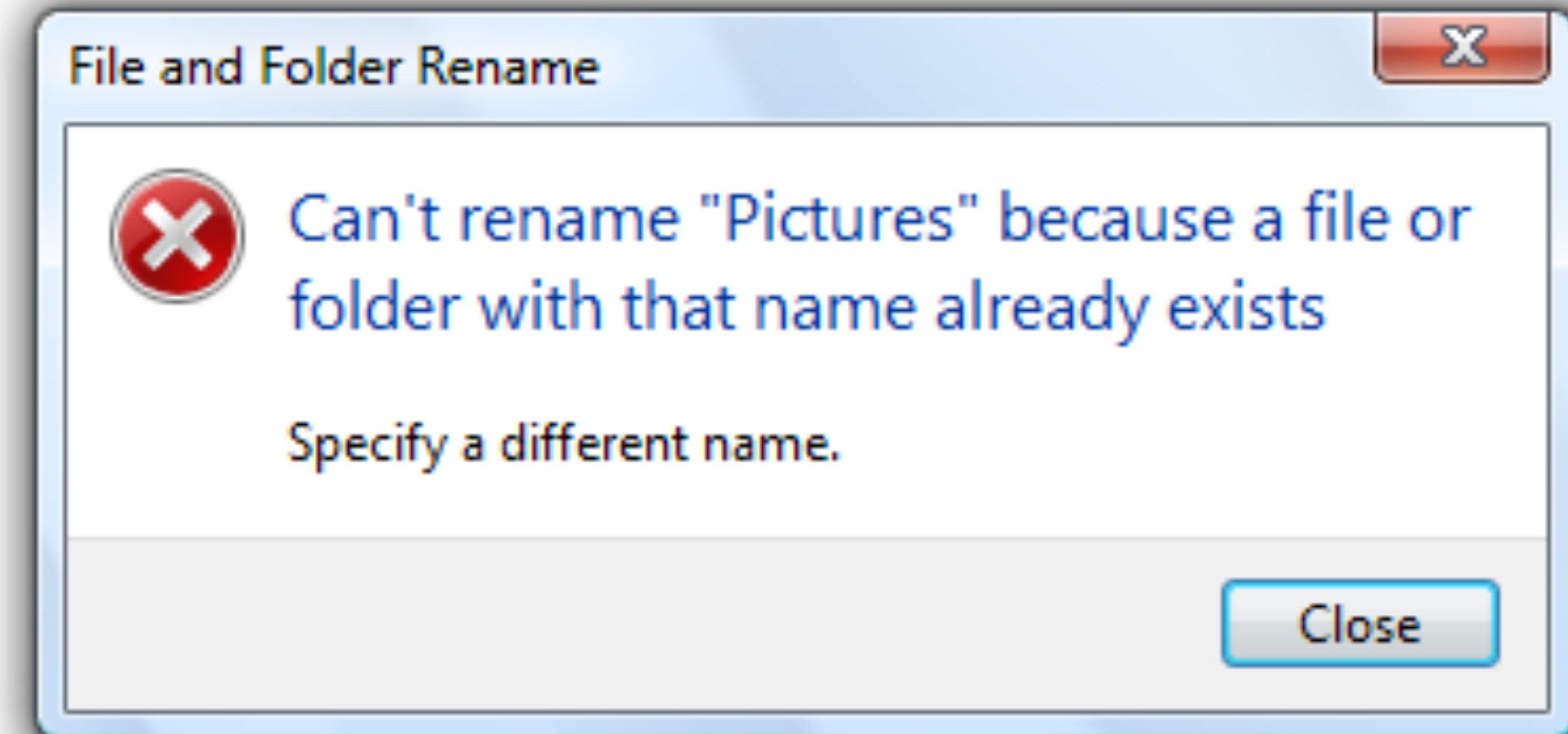
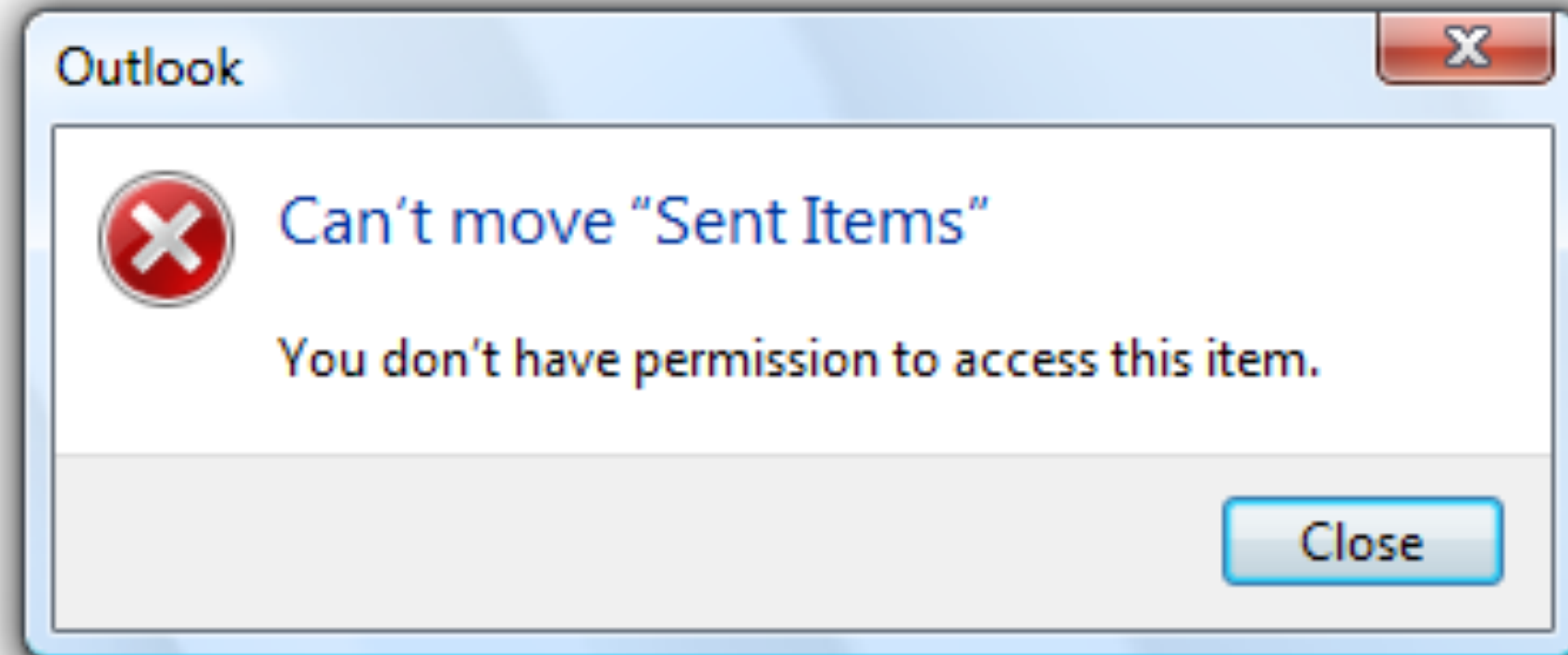
# User-Friendly Error Messages



# Bad Error Messages



# Good Error Messages



# Recursive Grep, using Scanner

## Cause: missing arguments

```
> java RecursiveGrepScanner
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds for length 0
 at RecursiveGrepScanner.main(RecursiveGrepScanner.java:41)
```

## Cause: missing dependencies, since java-introcs wasn't used

```
> java RecursiveGrepScanner dream poems
Exception in thread "main" java.lang.NoClassDefFoundError: StdOut
 at RecursiveGrepScanner.searchFile(RecursiveGrepScanner.java:18)
 at RecursiveGrepScanner.findFiles(RecursiveGrepScanner.java:35)
 at RecursiveGrepScanner.findFiles(RecursiveGrepScanner.java:32)
 at RecursiveGrepScanner.findFiles(RecursiveGrepScanner.java:32)
 at RecursiveGrepScanner.main(RecursiveGrepScanner.java:41)
Caused by: java.lang.ClassNotFoundException: StdOut
 at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:641)
 at java.base/jdk.internal.loader.ClassLoaders$AppClassLoader.loadClass(ClassLoaders.java:188)
 at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:521)
 ... 5 more
```

# Recursive Grep, using Scanner

## Cause: unreadable file

```
> java-introcs RecursiveGrepScanner dream poems
poems/project-gutenberg/alexander-pushkin/the-roussalka.txt The dreamy wave she vanished under.
Exception in thread "main" java.io.FileNotFoundException: poems/test.txt (Permission denied)
 at java.base/java.io.FileInputStream.open0(Native Method)
 at java.base/java.io.FileInputStream.open(FileInputStream.java:219)
 at java.base/java.io.FileInputStream.<init>(FileInputStream.java:158)
 at java.base/java.util.Scanner.<init>(Scanner.java:681)
 at java.base/java.util.Scanner.<init>(Scanner.java:659)
 at RecursiveGrepScanner.searchFile(RecursiveGrepScanner.java:14)
 at RecursiveGrepScanner.findFiles(RecursiveGrepScanner.java:35)
 at RecursiveGrepScanner.main(RecursiveGrepScanner.java:41)
```

# Refactoring Recursive Grep

- Don't crash when a particular file can't be read
  - Instead, print an error to STDERR
- Describe the required command-line arguments
  - Use an argument parsing library, such as Argparse4j



# After Refactoring

```
> java RecursiveGrepUsable
usage: RecursiveGrepUsable [-h] target directory
RecursiveGrepUsable: error: too few arguments
```

```
> java RecursiveGrepUsable -h
usage: RecursiveGrepUsable [-h] target directory
```

Recursively search a directory for files matching a target string

positional arguments:

|           |                              |
|-----------|------------------------------|
| target    | Search for this string       |
| directory | Search within this directory |

named arguments:

|            |                                 |
|------------|---------------------------------|
| -h, --help | show this help message and exit |
|------------|---------------------------------|

# After Refactoring

```
> java RecursiveGrepUsable dream poems
poems/project-gutenberg/alexander-pushkin/the-roussalka.txt The dreamy wave she vanished under.
poems/public-domain-poetry/charlotte-bronte/life.txt Life, believe, is not a dream
poems/public-domain-poetry/charlotte-bronte/the-wood.txt We'll seek a couch of dreamless ease;
poems/public-domain-poetry/edgar-allan-poe/the-raven.txt Doubting, dreaming dreams no mortals ever dare
poems/public-domain-poetry/edgar-allan-poe/the-raven.txt And his eyes have all the seeming of a demon's
Error reading 'poems/test.txt': java.io.FileNotFoundException: poems/test.txt (Permission denied)
```