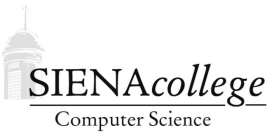

Computer Science 225
Advanced Programming
Siena College
Spring 2022

Topic Notes: Files and Exception Handling

You have been using file input and output in your Java programs almost since the beginning of your programming careers. You are most likely familiar with using the `Scanner` class to read files and the `PrintWriter` class to write them. We will be looking more carefully next at some of the issues that arise when dealing with files in Java.

The `File` class

Whichever of Java's mechanisms you have used to work with files, you have encountered the `java.io.File` class. However, it is unlikely that you did much more than to construct a `File` object and then pass it to the constructor of a `Scanner` or `PrintWriter`.

See the API documentation for this class to see all of the other features and functionality of `java.io.File`.

`https://github.com/SienaCSISAdvancedProgramming/FileMethods`

Try this out on some files that exist, files that don't exist, directories, etc.

File I/O with NIO.2

This is not a core topic of the course, but is important if you will be using Java to perform large amounts of file input and output.

You are experts on using the `File` class along with `Scanner` and `PrintWriter` to read and write files. Modern versions of Java have a more extensive API called NIO.2 for file and file system I/O. In particular, the classes `Path`, `Files`, and `FileSystem` in the `java.nio.file` package form the core of this system.

You can learn much more about Java's I/O features, including NIO.2 functionality, at `https://docs.oracle.com/javase/tutorial/essential/io/fileio.html`

Exceptions

A complication of working with files is that things can sometimes go wrong. A file might not exist, or you might not have the privileges to create a file in a given location. You have learned that in Java, such error conditions are often dealt with by throwing *exceptions*, since the part of the code that *detects* the error might not be able to *handle* the error in a manner that is appropriate to all situations.

When working with files, programmers sometimes just have their own `main` (or other) method

include a `throws` *clause* which passes the error to the caller. When done in a `main` method, this results in the program crashing and an error message and call stack trace being printed.

For the remainder of this section, refer to the examples in:

`https://github.com/SienaCSISAdvancedProgramming/CatchExamples`

It is usually better to handle the exception by placing the code that could throw an exception inside a `try` block, with an appropriate `catch` block following that defines an *exception handler* for that type of exception. A feature of exception handling that you might not have seen before is the `finally` clause, where you can place code that you want to execute even if an exception handler happens. This is demonstrated in `CatchExample.java`.

A better way to handle the closing of your files in the event of exceptions is to use the *try with resources* form of exception handling, as you can see in `CatchExampleWithResources.java`.

Any resource opened in the `try` which implements the `AutoClosable` interface (which requires an appropriate `close` method to be defined) will have that `close` method called as necessary.

User-Defined Exceptions

You can (and often should) design your own Java classes to throw exceptions when appropriate. By doing so, you allow the program that is using your class to handle the exception in a way that is appropriate to its usage. For example, if you have an implementation of a linked list where someone asks to get an item that does not exist, it is better to throw an exception rather than just print an error message or, worse yet, terminate the entire program.

In some cases, you can use one of the many exception types that are defined by the standard Java API. In other cases, you might define your own, as in this example that should be familiar to many of you:

`https://github.com/SienaCSISAdvancedProgramming/Matrix2D`