# Reading Files

# Files

- We've used data structures to manage all our data so far
  - Why ever use a file?

- Data structures are stored in memory
  - Only exist while our program is running

- Files are stored on the hard drive (disk)
  - Persistant storage

# Reading Files

- Let's read files in Java!
- We'll break down this example

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;

public static ArrayList<String> readFile(String filename) {
    return new ArrayList<>(Files.readAllLines(Paths.get(filename)));
}
```

# Reading Files

- Before reading a file
  - We need to create a Path object from the filename
- Paths needs to be imported so we can call it's get static method

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;

public static ArrayList<String> readFile(String filename) {
    return new ArrayList<>(Files.readAllLines(Paths.get(filename)));
}
```

# Reading Files

- Call the static method readAllLines from the imported Files object
- There are other ways to read files in Java
  - This is the simplest, but it is *unbuffered* an is inefficient for very large files (This method is fine for CSE116)

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;

public static ArrayList<String> readFile(String filename) {
    return new ArrayList<>(Files.readAllLines(Paths.get(filename)));
}
```

# Reading Files

- Files.readAllLines returns a List
  - If you don't want to work with the List interface, create a new ArrayList

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;

public static ArrayList<String> readFile(String filename) {
    return new ArrayList<>(Files.readAllLines(Paths.get(filename)));
}
```

# Exceptions

- This code is dangerous!
  - It relies on data that is outside the control of this program (The file)
  - If the file doesn't exist, this program *throws an exception*

```
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;

public static ArrayList<String> readFile(String filename) {
    return new ArrayList<>(Files.readAllLines(Paths.get(filename)));
}
```

# Exceptions

- Exceptions are thrown when something happens in a method call that they doesn't handle
- Commonly, you see throw exceptions when there's an error in your code
  - IndexOutOfBoundsException, NullPointerException, StackOverflowException, etc.
- An *uncaught* exception will crash the program
- We don't want our program to crash if the file is not found, so we will *catch* the exception

# Exceptions

- We can run risky code in a try block
- If an exception is thrown, run the code in the corresponding catch block

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;

public static ArrayList<String> readFile(String filename) {
    try {
        return new ArrayList<>(Files.readAllLines(Paths.get(filename)));
    } catch (IOException e) {
        return new ArrayList<>();
    }
}
```

# Exceptions

- We *try* to run this code
- It might thrown an exception
- if it does throw an exception, stop running code from this block and move to the catch block

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;

public static ArrayList<String> readFile(String filename) {
    try {
        return new ArrayList<>(Files.readAllLines(Paths.get(filename)));
    } catch (IOException e) {
        return new ArrayList<>();
    }
}
```

# Exceptions

- Write catch blocks for specific exception types
- We're expecting an IOException so we will catch this type of exception
- Can have multiple catch blocks if multiple types of exceptions can be thrown
- If an exception is throw that doesn't match the type of any catch block, the exception is uncaught and the program will crash

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;

public static ArrayList<String> readFile(String filename) {
    try {
        return new ArrayList<>(Files.readAllLines(Paths.get(filename)));
    } catch (IOException e) {
        return new ArrayList<>();
    }
}
```

# Exceptions

- If an IOException is thrown
- Run the code in the catch block
- This code will attempt to read the file
  - Return an ArrayList containing all the lines of the file
  - Return an empty ArrayList if the file does not exist

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.util.ArrayList;

public static ArrayList<String> readFile(String filename) {
    try {
        return new ArrayList<>(Files.readAllLines(Paths.get(filename)));
    } catch (IOException e) {
        return new ArrayList<>();
    }
}
```

# String Parsing

# String Parsing

- We'll read several CSV files in this course
- To handle CSV files, we'll split each line of the file on commas

- Note: This is not the proper way to parse CSV as the format is more complicated that this (eg. if the data itself contains commas). The data we'll handle in CSE116 is specifically prepared to allow splitting on commas

# String Splitting

- Use the split method of the String class to separate a String by a delimiter

- This example will split the String "a,b,c" on commas to return an Array with the values "a", "b", and "c"

```
String line = "a,b,c";
ArrayList<String> splits = new ArrayList<>(Arrays.asList(line.split(",")));
```

# String Splitting

- Split returns an Array
- If you don't want to use a plain Array
  - Convert the Array to a List using Arrays.asList
  - Create a new ArrayList with the List as a constructor argument

```
String line = "a,b,c";
ArrayList<String> splits = new ArrayList<>(Arrays.asList(line.split(",")));
```

# String to Number

- Sometimes we have to work with numbers that are stored in files as Strings
- Use the corresponding parse method to convert the String to an int or a double
- These methods only work if the number is "well-formed" meaning that it can be converted
  - eg. parsing the String "four" as an int will throw an exception

```
int anInt = Integer.parseInt("64");
double aDouble = Double.parseDouble("1.5");
```

# Live Coding