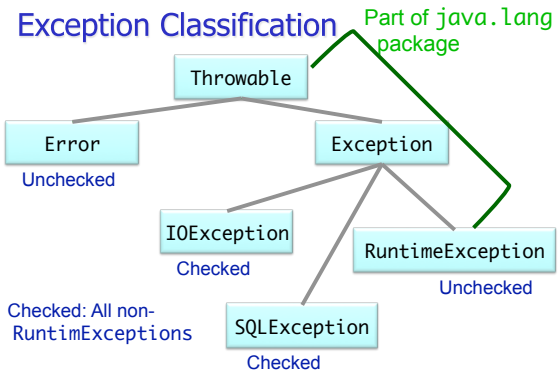
Objectives

- Exceptions
- Files
- Streams

Review

- Permissions
  - What are the categories who permissions are applied to?
  - What types of permissions can be granted?
- Exceptions
  - What are the two types of exceptions?
  - What is one way to deal with exceptions?

Exception Classification



Factorial Alternatives

```
public static double factorial( int x ) {
    if( x < 0 )
       return 0.0;
    double fact = 1.0;
    while( x > 1 ) {
       fact *= x;
       x--;
    }
    return fact;
}
```


Factorial Alternatives

```
public static double factorial( int x ) {
    if( x < 0 )
       throw new IllegalArgumentException("x
 must be >= 0");
    double fact = 1.0;
    while( x > 1 ) {
       fact *= x;
       x--;
    }
    return fact;
}
```

Could also use `assert` More later…

What are the pros and cons of these approaches?

A More Descriptive Exception

- Four constructors for **all** Exception classes
  - Default (no parameters)
  - Takes a `String message`
    - Describe the condition that generated this exception more fully
  - 2 more

```
if (num_read < num_bytes) {
    String problem = "I read " + num_read +
       " when I should have read " + num_bytes;
    throw new EOFException(problem);
}
```

Creating Our Own Exception Class

- The `EOFException` class described the error our method encountered well
  - Not always the case
  - Many exceptions derived from `IOException` but plenty more conditions
- If you cannot find a predefined Java `Exception` class that describes your condition, make a new `Exception` class!

Creating Our Own Exception Class

```
public class FileFormatException extends IOException {
      public FileFormatException()
      {
      }

      public FileFormatException(String gripe) {
           super(gripe);
      }
}
```

What happens in this method implicitly?

- Can now throw exceptions of type `FileFormatException`

Catching Exceptions

- After we throw an exception, some part of our program needs to *catch* it
  - Knows how to deal with the situation that caused the exception
  - Receives it
  - Handles the problem
    - Hopefully gracefully, without exiting

The try/catch Block

- The simplest way to catch an exception is to use a *try/catch* block
- Simplest form of this block looks like:

```
try {
      code;
      more code;
}
catch (ExceptionType e) {
      error code for ExceptionType;
}
```


Try/Catch Block

```
try {
      code;
      more code;
}
catch (ExceptionType e) {
      error code for ExceptionType;
}
```

- The code in the `try` block runs first
  - If `try` block completes without an exception, `catch` block(s) are skipped
  - If the `try` code generates an exception, a `catch` block runs
    - Remaining code in the `try` block is skipped

The try/catch Block

```
try {
      code;
      more code;
}
catch (ExceptionType e) {
      error code for
      ExceptionType
}
```

- If code inside `try` block does *not* throw an exception of `ExceptionType`, `catch` block is skipped
- If an exception of a type other than `ExceptionType` is thrown inside `try` block, method exits immediately and the program dies

The try/catch Block

```
try {
      code;
      more code;
}
catch (ExceptionType e) {
      error code for
      ExceptionType
}
catch (ExceptionType2 e) {
      error code
      for ExceptionType2
}
```

- You can have more than one `catch` block
  - To handle > 1 type of exception
- If `ExceptionType1` does not catch exception, falls to `ExceptionType2`, and so forth
  - Run the first matching `catch` block

Can catch any type with **`Exception e`** but won't have customized messages

Try/Catch Example

```
public void read(BufferedReader in) {
      try {
            boolean done = false;
            while (!done) {
                  String line=in.readLine();
                  // this could throw IOException!
                  if (line == null)
                        done = true;
            }
      }
      catch (IOException exp) {
            exp.printStackTrace();
      }
}
```

Prints out stack trace to method call that caused the error

Try/Catch Example

```
public void read(BufferedReader in) {
      try {
            boolean done = false;
            while (!done) {
                  String line=in.readLine();
                  // this could throw IOException!
                  if (line == null)
                        done = true;
            }
      }
      catch (IOException exp) {
            exp.printStackTrace();
      }
}
```

More precise try/catch may help pinpoint error
But could result in messier code

Stack Trace Example

```
java.io.FileNotFoundException: fred.txt
   at java.io.FileInputStream.<init>(FileInputStream.java)
   at java.io.FileInputStream.<init>(FileInputStream.java)
   at ExTest.readMyFile(ExTest.java:19)
   at ExTest.main(ExTest.java:7)
```

How helpful is this output?
How user friendly is it?

Stack Trace Example

```
java.io.FileNotFoundException: fred.txt
   at java.io.FileInputStream.<init>(FileInputStream.java)
   at java.io.FileInputStream.<init>(FileInputStream.java)
   at ExTest.readMyFile(ExTest.java:19)
   at ExTest.main(ExTest.java:7)
```

How helpful is this output?
How user friendly is it?

- Useful for debugging your code
- Generate/display user-friendly errors in finished product

The `finally` Block

- Can add a `finally` block after all possible `catch` blocks
  - Code in `finally` block **always** runs after the code in `try` and/or `catch` blocks
    - After `try` block finishes; or if an exception occurs, after the `catch` block finishes
- Allows you to clean up or do maintenance before the method ends (one way or the other)
  - E.g., closing files or database connections

Practice: try/catch/finally Blocks

```
try {
      statement1;
      statement2;
}
catch (EOFException e) {
      statement3;
      statement4;
}
finally {
      statement5;
}
```

- Which statements run if:
  - Neither `statement1` nor `statement2` throws an exception
  - `statement1` throws an `EOFException`
  - `statement2` throws an `EOFException`
  - `statement1` throws an `IOException`

What to do with a Caught Exception?

- We dump the stack after the exception occurs
  - What else can we do?
- Often, the best answer is to do nothing but report the problem
- If an exception occurs in the `readLine()` method, our `readData()` method should probably pass up to whoever called it
- Instead of catching this exception, advertise that the `readData()` method can throw an IOException
  - Let whoever calls the `readData()` method catch and handle the exception

Methods Throwing Exceptions

- API documentation tells you if a method can throw an exception
  - If so, you must handle it
- If your method could possibly throw an exception (by generating it or by calling another method that could), advertise it!
  - If you can't handle every error, that's OK…let whoever is calling you worry about it
  - However, they can only do that if you advertise the exceptions you can't deal with

Programming with Exceptions

- Exception handling is slow
- Use one big `try` block instead of nesting `try -catch` blocks too deep
- Don't ignore exceptions (e.g., `catch` block does nothing)
  - Better to pass them along to higher calls

Benefits of exceptions?

Benefits of Exceptions

- Force error checking/handling
  - Otherwise, won't compile
  - Does not guarantee "good" exception handling
- Ease debugging
  - Stack trace

FILES

java.io.File Class

- Represents a file on the disk
- Provides functionality such as
  - Storage of the file on the disk
  - Determine if a particular file exists
  - When file was last modified
  - Rename file
  - Remove/delete file
  - …

Making a File Object

- Simplest constructor takes full file name (including path)
  - If don't supply path, Java assumes current directory (.)

```
File f1 = new File("chicken.data");
```

  - Creates a File *object* representing a file named "chicken.data" in the current directory
  - Does ***not*** create a file with this name on disk

Making a File Object

- File object represents a file with that pathname on the disk
  - Even if file does not exist
- File's `exists()` method
  - Determines if a file exists on the disk
  - Create a File object that represents file and call `exists()` method.

Files and Directories

- A File object can represent a file **or** a directory
  - Directories are special files in most modern operating systems
- Use `isDirectory()` and/or `isFile()` to see what type of file is File object represents

More File Constructors

- String for the path, String for filename

```
File f2 = new File(
   "/home/courses/cs209/datafiles","chicken.data");
```

- File for directory, String for filename

```
File dir= new File("/home/courses/cs209/datafiles");
File f4 = new File(dir, "chicken.data");
```

"Break" any of Java's Principles?

Not Portable

- Accessing the file system is inherently not portable
  - In Windows, paths are "c:\\dir"
  - In Unix, paths are "/home/courses/dir"
- Relies on underlying file system/operating system to perform actions

Handling Portability Issues

- Fields in File class
  - `static separator`
    - Unix: "/"
    - Windows: "\\"
  - `static pathSeparator`
    - For separating a list of paths
    - Unix: ":"
    - Windows: ";"
- Use relative paths, with separators

Why two \\?

`java.io.File` Class

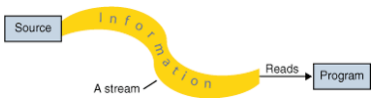
- 25+ methods
  - Manipulate files and directories
  - Creating and removing directories
  - Making, renaming, and deleting files
  - Information about file (size, last modified)
  - Creating temporary files
  - …
- See online API documentation

`FileTest.java`

STREAMS

Streams

- Java handles input/output using **streams**, which are sequences of bytes



input stream: an object from which we can *read* a sequence of bytes

Abstract class: `java.io.InputStream`

Streams

- Java handles input/output using **streams**, which are sequences of bytes

output stream: an object to which we can write a sequence of bytes

Abstract class: `java.io.OutputStream`

Console I/O

- Output:
  - `System.out`, which is a `PrintStream` object
- Input
  - `System.in` is an `InputStream`
  - Throws exceptions if format of input data is not correct
    - Handle in `try/catch`

Java Streams

- MANY (80+) types of Java streams
- In `java.io` package
- Why streams?
  - Information stored in different sources is accessed in essentially the same way
    - Example sources: file, on a web server across the network, string
  - Allows same methods to read or write data, regardless of its source
    - Create an `InputStream` or `OutputStream` of the appropriate type

Opening & Closing Streams

- Streams are *automatically opened* when created
- Close a stream by calling its `close`() method
  - Close a stream as soon as object is done with it
  - Free up system resources

Reading & Writing Bytes

- Parent class: `InputStream`
  - `abstract int read()`: reads one byte from the stream and returns it
- Concrete input stream classes override `read`() to provide appropriate functionality
  - e.g., FileInputStream class: read() reads one byte from a file
- Similarly, `OutputStream` class has abstract `write`() to write a byte to the stream

Reading & Writing Bytes

- `read`() and `write`() are ***blocking*** operations
  - If a byte cannot be read from the stream, the method waits (does not return) until a byte is read
- `isAvailable`() allows you to check the number of bytes that are available for reading before you call `read`()

```
int bytesAvailable = System.in.isAvailable();
if (bytesAvailable > 0)
      System.in.read(byteBuffer);
```

More Powerful Stream Objects

- `DataInputStream` class
  - Directly reads Java primitive types through method calls such as `readDouble()`, `readChar()`, `readBoolean()`
- `DataOutputStream` class
  - Directly writes Java primitive types with `writeDouble()`, `writeChar()`, …

File Input and Output Streams

- `FileInputStream`: provides an input stream that can read from a file
  - Constructor takes the name of the file:

```
FileInputStream fin = new
          FileInputStream("chicken.data");
```

  - Or, uses a `File` object …

```
File inputFile = new File("chicken.data");
FileInputStream fin = new FileInputStream(inputFile);
```

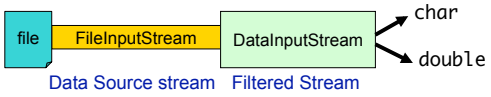
FileTest.java

Filtered Streams

- `FileInputStream` has no methods to read numeric types
- `DataInputStream` has no methods to read from a file
- Java allows you to **combine** two types of streams into a **connected stream**

Connected Streams

- Think of a stream as a "pipe"
- `FileInputStream` knows how to read from a file
- `DataInputStream` knows how to read an `InputStream` into useful types
- Connect the **out** end of the FileInputStream to the **in** end of the DataInputStream…



DataIODemo.java

Filtered Streams vs Data Source Streams

- Subclasses of `FilterInputStream` or `FilterOutputStream`
- Always contains another stream
- Adds functionality to other stream
  - Automatically buffered IO
  - Automatic compression
  - Automatic encryption
  - Automatic conversion between objects and bytes
- As opposed to Data source streams
  - communicate with a data source
    - file, byte array, network socket, or URL

Filtered Streams: Reading from a file

- If we wanted to read numbers from a file
  - `FileInputStream` reads bytes from file
  - `DataInputStream` handles numeric type reading
- Connect the `DataInputStream` to the `FileInputStream`
  - `FileInputStream` gets the bytes from the file and `DataInputStream` reads them as assembled types

```
FileInputStream fin = new
          FileInputStream("chicken.data");
DataInputStream din = new
          DataInputStream(fin);   "wrap" fin in din
double num1 = din.readDouble();
```

Aside: StringBuffers vs Strings

- Strings are "read-only" or immutable
- Use StringBuffer to manipulate a String
- Instead of creating a new String using
  - `String str = prevStr + " more!";`
- Use

```
StringBuffer str = new StringBuffer( prevStr );
str.append(" more!");
```

- Many StringBuffer methods, including toString() to get the resultant string back

Buffered Streams

- Use a BufferedInputStream class object to buffer your input streams
  - A pipe in the chain that adds buffering
  - Speeds up access

```
DataInputStream din = new DataInputStream (
   new BufferedInputStream (
      new FileInputStream("chicken.data")));
```


A More Connected Stream

- FileInputStream reads bytes from the file
- BufferedInputStream buffers bytes
  - speeds up access to the file
- DataInputStream reads buffered bytes as types