Exceptions

- Objectives - when we have completed this set of notes, you should be familiar with:
  - the purpose of exceptions
  - exception message and call stack trace
  - try-catch-finally blocks
  - propagating exceptions
  - checked and unchecked exceptions
  - throw statement and creating exceptions
  - I/O Exceptions – using try-catch blocks for File I/O
  - reading and writing text files
  - opening files in the default web browser

Exceptions

- An ***exception*** is an object that describes an unusual or erroneous situation
- Exceptions are ***thrown*** (or raised) by a program during execution; they may be ***caught*** and ***handled***, or they may be ignored (as we've been doing)
- A program can be separated into normal execution flow and ***exception execution flow***
- An ***error*** is also represented as an object in Java, but usually represents an unrecoverable situation and should not be caught

Exceptions

- If an exception is ignored by the program, a run-time error will occur and the exception and *exception message* (optional) and *call stack trace* will be printed

  An example you've likely seen:

```
----jGRASP exec: java Test

Exception in thread "main" java.lang.NullPointerException
        at Test.main(Test.java:6)

----jGRASP wedge2: exit code for process is 1.
----jGRASP: operation complete.
```

- The call stack trace shows the method call trail that led to the attempted execution of the offending line of code
- See Zero1.java

Exception Handling

- Java has a predefined set of exceptions and errors that can occur during execution
  Examples:
  - ArrayIndexOutOfBoundsException in the java.lang package
  - NumberFormatException in the java.lang package
- A program can deal with an exception in one of three ways:
  - ignore it
  - handle it where it occurs
  - handle it an another place in the program

try-catch Blocks

- To process an exception where it occurs, the statement that throws (or raises) the exception is executed within a *try block*
- A try block is usually followed by one or more *catch* blocks that specify the exception(s) to be caught and handled. A try block must be followed by a catch or finally (unless it's a try-with-resources, which will not be covered)
- When an exception occurs, processing continues at the first catch block that matches the exception type

`Zero2.java` `AbsoluteValue1.java` `AbsoluteValue2.java`

The *finally* Block

- A try block can be followed by a `finally` block
- Once a program enters the try block, the statements in the finally block are always executed [unless System.exit() is called]
  - If no exception is generated, then **after the statements in the try block complete**, the statements in the **finally** block are executed
  - If an exception occurs, control jumps to the matching **catch** block, if any, and its statements are executed, and then the statements in the **finally** block are executed
- See `Zero3.java` `Zero4.java` `GuessNumber1.java` `GuessNumber2.java`

Exception Propagation

- An exception can be handled at a higher level if it is not appropriate to handle it where it occurs
- An exception *propagates* up through the method calling hierarchy until it is caught and handled or if it reaches the `main` method and is still not caught, the program ends abnormally
- A try block that contains a call to a method in which an exception is thrown can be used to catch that exception
- See `Propagation.java` `ExceptionScope.java`

Checked Exceptions

- An exception is either *checked* or *unchecked*
- A *checked exception* either must be caught by a method, or must be listed in the *throws clause* of any method that may throw or propagate it
- A throws clause is appended to the method header (e.g., throws FileNotFoundException)
- The compiler will issue an error if a checked exception is not handled appropriately

Unchecked Exceptions

- An unchecked exception does not require you to handle it (e.g., you have likely encountered a NullPointerException without having try-catch blocks or a throws clause for NullPointerException)
- Unchecked exceptions in Java are objects of type `RuntimeException` and its descendants (see Java API for Exception, RuntimeException, etc.)
- Error objects are similar to `RuntimeException` objects in that they are unchecked
  - Errors do not require a throws clause
  - Errors should not be caught (e.g., OutOfMemoryError)

The throw Statement

- You may want to explicitly throw an exception in a method
  - Often better than ignoring input / actions / etc. that your program has detected as incorrect
- Exceptions are thrown using the *throw* statement
- Usually an if statement evaluates a condition(s) to see if the exception should be thrown
- You can create your own exception if there is not an appropriate exception in the Java API See `PolygonCreator.java`

I/O Exceptions

- A *stream* is a sequence of bytes that flows from a source to a destination
- In a program, we read information from an input stream and write information to an output stream
- A program can manage multiple I/O streams simultaneously

I/O Exceptions

- There are three standard I/O streams:
  - *standard input* – defined by `System.in`
  - *standard output* – defined by `System.out`
  - *standard error* – defined by `System.err`
- `System.in` is typically keyboard input
  - We've been using the Scanner class to read from System.in
- `System.out` and `System.err` are typically shown in a particular window on the screen
  - We use `System.out` when we execute `println` statements

I/O Exceptions

- The `java.io` package contains many classes that allow us to define various streams with particular characteristics
- Some classes assume that the data consists of characters, which is our focus
- Others assume that the data consists of raw bytes of binary information
- Many of the I/O classes can potentially throw an IOException or one of its subclasses (e.g., FileNotFoundException)
- IOException is a checked exception

I/O Exceptions

- Now we want to consider how to read and write files and how to handle I/O exceptions
- For reading from a file, we use the following:
  - java.io.File and java.util.Scanner
- For writing to a file, we use the following:
  - java.io.PrintWriter

Reading from a File

- To read from a text file, you can create an instance of the File class in java.io, using the name of the file to be read (a String)
- You can then instantiate a Scanner object using the File object that you created
- At that point, you can use the Scanner methods to read the file (and these should familiar to you):
  - The next method reads a "token"
  - The nextLine method reads a whole line
  - The hasNext and hasNextLine are also useful (see API documentation for more information and methods)

Reading from a File

- The Scanner class has a constructor that accepts a File object as a parameter (see Java API), which throws FileNotFoundException, a subclass of IOException, so you can catch either
- FileNotFoundException is a checked exception, so you have to do one of two things …
  - Specify throws FileNotFoundException in the method header
  - Handle the exception with a try-catch; good practice to close the Scanner object for a file (consider finally block)
- See ReadLines1.java ReadLines2.java ReadLines3.java ReadLines3.java

Writing to a File

- Instantiate a PrintWriter object using the file name (a String)
  - The PrintWriter constructor throws FileNotFoundException
- PrintWriter has methods similar to System.out
  - print: writes a specified String to a file
  - println: writes a specified String and a new line to a file
  - **When writing is completed, invoke the close() method on the PrintWriter object to flush buffer; otherwise, nothing may be written to the file!**
- See WriteLines1.java WriteLines2.java WriteReadRandom.java

Writing HTML to a File

- Similar to writing plain text with PrintWriter
- Add HTML tags to the text (here are a few)
  - Heading <h1> . . . </h1>
  - Paragraph . . .
  - Line break
  - Bold . . .
  - Pre-format <pre> . . . </pre>
  - Font color . . .
- Opening HTML file in default browser
- See WriteLinesHTML.java WriteRandomHTML.java