File I/O and Exceptions

- File I/O
- Exceptions
- Throwing Exceptions
- Try statement and catch / finally clauses
- Checked and unchecked exceptions
- Throws clause
- Reading for this lecture: L&L 10.8, 11.1 – 11.6

CLI File Input

- In a CLI, we want the user to select a file within a directory system so that its contents can be read and processed
- However, we must rely on the user typing in the file name (including any required path name)
- We can get the file name via a Scanner on `System.in` using the `nextLine` method
- We can read the file data via a Scanner on a `File` object using the `nextLine` method again

CLI File Input: Example

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

public class FileDisplay
{
  public static void main (String [] args)
                          throws IOException
  {
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter name of file to display");
    File file = new File(scan.nextLine());

    scan = new Scanner (file);  // done with keyboard
    while (scan.hasNext())      // ctl-D returns false
      System.out.println(scan.nextLine());
  }
}
```

CLI File Output

- In a CLI, we want the user to create a file within a directory system so that its contents can be written (or overwritten!)
- Be careful: Your code should check for a file by that name and ask user if OK to overwrite it.
- Again, we rely on the user typing in the file name
- Again, we can get the file name via a Scanner on `System.in` using the `nextLine` method
- We can write the file data via a `PrintStream` on a `File` object using the `println` method (`System.out` is a `PrintStream` object)

CLI File Output: Example

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

public class FileWrite
{
  public static void main (String [] args) throws IOException
  {
    // Get filename and instantiate File object as before

    PrintStream out = new PrintStream(file);
    // Use ctl-D to close System.in
    // so scan.hasNext() will return false
    while (scan.hasNext()) {
      String line = scan.nextLine();
      out.println(line);
    }
    out.close();
  }
}
```

GUI File I/O

- In a GUI, requiring the user to enter a file name (including a path name or not) is considered to be NOT very user friendly
- We want our program to offer a choice of the available files so that the user can:
  – Move around within the available directories
  – Select one of the files shown in a directory

File Chooser in GUI’s

- Recall that a dialog box is a small window that "pops up" to interact with the user for a brief, specific purpose
- A *file chooser*, the `JFileChooser` class, supports a simple dialog box for this process
- See `DisplayFile.java` (page 521)

Example: DisplayFile code segment

```
JFileChooser chooser = new JFileChooser();

int status = chooser.showOpenDialog(frame);
//   There is also a showSaveDialog(frame)

if (status != JFileChooser.APPROVE_OPTION)
   ta.setText ("No File Chosen");
else
{  // read file
   File file = chooser.getSelectedFile();
   Scanner scan = new Scanner (file);
   ...
```

File Input/Output

- Notice that the main method in all three of these examples indicates that the code may `throw` an `IOException`
- If an error such as “file not found” occurs during a file operation, an `IOException` is generated by the system
- We’ll study exceptions in the next lecture

Exceptions

- An *exception* is an object that flags/ describes the occurrence of an unusual or erroneous situation
- Java has a predefined set of Exception classes for errors that can occur during execution
  - e.g ArithmeticException
- We can write our own Exception classes if needed
- When code in a program detects an “impossible condition”, it can *throw* a defined exception object
- The manner in which exceptions are processed is an important design consideration

Throwing Exceptions

- For code to “throw” an exception:
  - It must detect the “impossible” situation
  - Instantiate and “throw” an exception object
- Example (throw is a Java reserved word):

```
if (boolean logic to detect impossible situation)
      throw new NameOfException(“text to print”);
```

- Some Java statements or methods in the class library may throw exceptions this way

Handling Exceptions

- A program can deal with an exception in one of three ways:
  - ignore it (Let the JVM shut down the program)
  - handle it where it occurs
  - handle it at another place in the program
- If we ignore it, we get something like this in the interactions pane (See Zero.java):

```
java.lang.ArithmeticException: / by zero
    at Zero.main(Zero.java:17)
    at sun.reflect.NativeMethodAccessor…
    …
```

The `try` Statement / `catch` Clause

- To handle an exception in a program, the line that may throw the exception is executed within a `try` statement followed by one or more `catch` clauses
- Each `catch` clause has an exception type and reference name and is called an *exception handler*
- If an exception occurs,
  - Processing stops in the body of the try statement
  - Processing continues at the start of the first `catch` clause matching the type of exception that occurred
- The reference name can be used in the `catch` clause to get information about the exception

The `finally` Clause

- A try statement can have an optional clause following the catch clauses, designated by the reserved word `finally`
- The Java statements in the finally clause are always executed
  - If no exception is generated, the statements in the finally clause are executed after the statements in the try block complete
  - If an exception is generated, the statements in the finally clause are executed after the statements in the appropriate catch clause complete

Example of `try-catch-finally`

```
try
{
  System.out.println(Integer.parseInt(string));
}
catch (NumberFormatException e)
{
  System.out.println("Caught exception: " + e);
}
finally
{
  System.out.println("Done.");
}
```

Exception Propagation

- An exception can be propagated up to the caller to be handled at a higher level if it is not appropriate to handle it where it occurs
- Exceptions *propagate* up through the method calling hierarchy until they are caught and handled or until they reach the level of the `main` method and/or JVM
- See `Propagation.java` (page 546)
- See `ExceptionScope.java` (page 547)

Checked/Unchecked Exceptions

- An exception is considered to be either *checked* or *unchecked*
- A `RunTimeException` or its decendents such as `ArithmeticException`, `NullPointerException`, etc are the only ones considered to be *unchecked*
- All other exceptions are considered to be *checked*
- Many of the checked exceptions are related to input / output, e.g. `IOException`

Checked Exceptions

- If a method can generate a checked exception, it must have a `throws` clause in its header
- (Note: "`throws`" is a different reserved word)
- If method1 calls method2 that has a `throws` clause in its method header, method1 must:
  - Use `try-catch` around the call to method2

    OR
  - Have a `throws` clause in its own method header
- The compiler will issue an error if a checked exception is not caught or listed in a `throws` clause

Example of the `throws` clause

```
public class FileDisplay
{
  public FileDisplay() throws IOException
  {
    Scanner scan = new Scanner(System.in);
    System.out.println("Enter name of file");
    File file = new File(scan.nextLine());

    // this line may throw an IOException
    // and its not inside a try statement
    scan = new Scanner (file);
```

Unchecked Exceptions

- An unchecked exception does not require explicit handling
- Code or calls to a method that may generate an unchecked exception can be put inside a `try-catch` statement, but that is optional