Notes

Paul's Thursday office hours will be moved to Friday—this week only

This is a LONG lecture.

CS112 –
Java
Programming

Spring 2024

Exceptions and File I/O

Exceptions

An *exception* is an object that describes an unusual or error situation

Exceptions are *thrown* by a program, and may be *caught* and *handled* by another part of the program

A program can be separated into a normal execution flow and an *exception execution flow*

Exception Handling

The Java API has a predefined set of exceptions that can occur during execution. Later we will define our own

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

The manner in which exceptions are processed is important in your software design

It is sometimes tedious to put error handling all over our code, after every keyboard read, math divide, etc. Exceptions give us a way to handle multiple errors all in one place.

Exception Handling

If an exception is ignored (not caught) by the program, the program will terminate and produce an appropriate message

The message may include a *call stack trace* that:

- indicates the line on which the exception occurred
- shows the method call trail that lead to the attempted execution of the offending line

See `Zero.java`

Give an example: main() calls SphereInfo.volume(), which calls radius(). Each function call is one entry in "the stack".

Output (when program terminates)

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Zero.main(Zero.java:17)
```

```
public class Zero
{
   //-----------------------------------------------------------------
   //  Deliberately divides by zero to produce an exception.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      int numerator = 10;
      int denominator = 0;

      System.out.println(numerator / denominator);

      System.out.println("This text will not be printed.");
   }
}
```

The try Statement

To handle an exception in a program, use a *try-catch statement*

A *try block* is followed by one or more *catch* clauses

Each catch clause has an associated exception type and is called an *exception handler*

When an exception occurs within the `try` block, processing immediately jumps to the first catch clause that matches the exception type

"try block" is a block of code that might raise an exception that you want to handle somehow

MODIFY ZEROS.JAVA IN ECLIPSE

The finally Clause

A `try` statement can have an optional `finally` clause, which is always executed

If no exception is generated, the statements in the `finally` clause are executed after the statements in the `try` block finish

If an exception is generated, the statements in the `finally` clause are executed after the statements in the appropriate `catch` clause finish

Exception Propagation

An exception can be handled at a higher level of your program if it is not appropriate to handle it where it occurs.

Can consolidate all error handling in one place!

Exceptions *propagate* up through the method calling hierarchy until they are caught and handled or until they reach the level of the `main` method

See `Propagation.java`

See `ExceptionScope.java`

```
//********************************************************************
//  Propagation.java       Author: Lewis/Loftus
//
//  Demonstrates exception propagation.
//********************************************************************

public class Propagation
{
   //-----------------------------------------------------------------
   //  Invokes the level1 method to begin the exception demonstration.
   //-----------------------------------------------------------------
   static public void main(String[] args)
   {
      ExceptionScope demo = new ExceptionScope();

      System.out.println("Program beginning.");
      demo.level1();
      System.out.println("Program ending.");
   }
}
```

```
//********************************************************************
//  ExceptionScope.java       Author: Lewis/Loftus
//
//  Demonstrates exception propagation.
//********************************************************************

public class ExceptionScope
{
   //-----------------------------------------------------------------
   //  Catches and handles the exception that is thrown in level3.
   //-----------------------------------------------------------------
   public void level1()
   {
      System.out.println("Level 1 beginning.");

      try
      {
         level2();
      }
      catch (ArithmeticException problem)
      {
         System.out.println();
         System.out.println("The exception message is: " +
                            problem.getMessage());
         System.out.println();
```

continue

continue

```
            System.out.println("The call stack trace:");
            problem.printStackTrace();
            System.out.println();
         }

         System.out.println("Level 1 ending.");
      }

      //-----------------------------------------------------------------
      //  Serves as an intermediate level.  The exception propagates
      //  through this method back to level1.
      //-----------------------------------------------------------------
      public void level2()
      {
         System.out.println("Level 2 beginning.");
         level3();
         System.out.println("Level 2 ending.");
      }
```

continue

continue

```
   //-----------------------------------------------------------------
   //  Performs a calculation to produce an exception.  It is not
   //  caught and handled at this level.
   //-----------------------------------------------------------------
   public void level3()
   {
      int numerator = 10, denominator = 0;

      System.out.println("Level 3 beginning.");
      int result = numerator / denominator;
      System.out.println("Level 3 ending.");
   }
}
```

Output

```
Program beginning.
Level 1 beginning.
Level 2 beginning.
Level 3 beginning.

The exception message is: / by zero

The call stack trace:
java.lang.ArithmeticException: / by zero
    at ExceptionScope.level3(ExceptionScope.java:
    at ExceptionScope.level2(ExceptionScope.java:
    at ExceptionScope.level1(ExceptionScope.java:
    at Propagation.main(Propagation.java:17)

Level 1 ending.
Program ending.
```

See that "Level 2 ending." and "Level 3 ending." are not printed.

Since those levels do not catch and handle the ArithmeticException, their normal processing is skipped.

Program control "passes up" until an exception handler is found.

And if no exception handler is found, the program exits.

```
    }
}
```

The `throws` clause

When defining a method that may throw exceptions, the method declaration includes a `throws` clause to list the exception types.

This lets users of the method decide how they plan to handle the Exceptions.

```
public int ReturnInventory(String partNum)
     throws
          IndexOutOfBoundsException,
          ArithmeticException
{
  // the code
}
```


There are many exception types that must be declared and some that do not need this. I can never keep track of which don't need to be mentioned, and just mention any I know about or that the compiler complains about.

The throw Statement

Exceptions are thrown using the *throw* statement

Usually a `throw` statement is executed inside an `if()` statement that evaluates a condition to see if the exception should be thrown

See `CreatingExceptions.java`

```
//********************************************************************
//  CreatingExceptions.java       Author: Lewis/Loftus
//
//  Demonstrates the ability to define an exception via inheritance.
//********************************************************************

import java.util.Scanner;

public class CreatingExceptions
{
   //-----------------------------------------------------------------
   //  Creates an exception object and possibly throws it.
   //-----------------------------------------------------------------
   public static void main(String[] args) throws OutOfRangeException
   {
      final int MIN = 25, MAX = 40;

      Scanner scan = new Scanner(System.in);
```

continue

continue

```
      System.out.print("Enter an integer value between " + MIN +
                       " and " + MAX + ", inclusive: ");
      int value = scan.nextInt();

      //  Determine if the exception should be thrown
      if (value < MIN || value > MAX) {
         throw new OutOfRangeException("Input value is out of range.");
       }
      System.out.println("End of main method.");  // may never reach
   }
}
```

Sample Run

```
Enter an integer value between 25 and 40, inclusive: 69
Exception in thread "main" OutOfRangeException:
    Input value is out of range.
    at CreatingExceptions.main(CreatingExceptions.java:20)
```

```
      if (value < MIN || value > MAX)
         throw problem;

      System.out.println("End of main method.");  // may never reach
   }
}
```

Quick Check

What is the matter with this code?

```
System.out.println("Before throw");
throw new OutOfRangeException("Too High");
System.out.println("After throw");
```

The throw is not conditional and therefore always occurs. The second `println` statement can never be reached.

Common Exception Types

IndexOutOfBoundsException
- Read an array or String index past the beginning or end

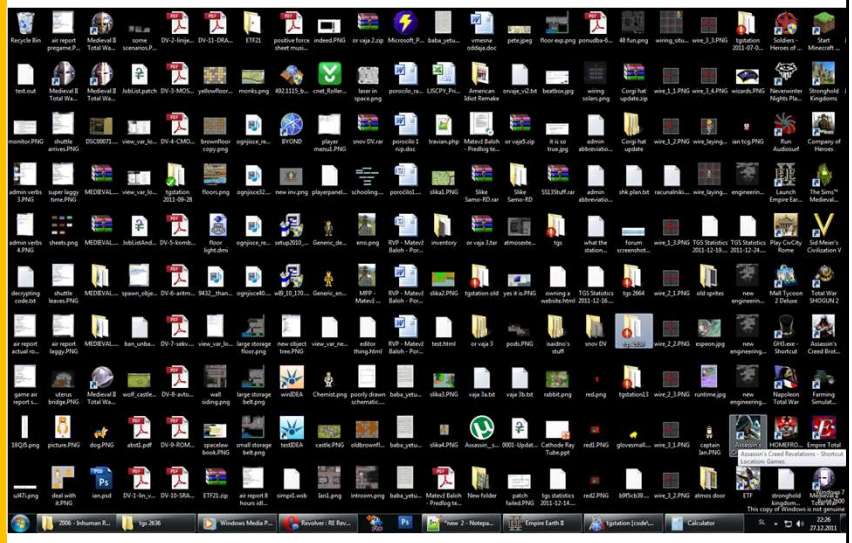
ArithmeticException
- Divide by 0, etc

NullPointerException
- Try to access a reference whose value is `null`

InputMismatchException
- Scanner throws this e.g. if try to read a double and input is not a number

In a few weeks we will learn about creating our own Exception types

File Input/Output



File I/O

So far we have read inputs from the command line or the keyboard

Now we will work on reading and writing **Files** and **Streams**

Files

File = data stored (readable, writable, modifiable, deletable) on persistent storage

A directory stores files *and* subdirectories

- Organized into hierarchical "tree" filesystem. Top is called "root directory"

```
/
        /users
                /users/phaskell
                        /users/phaskell/CS112
                /users/stephcurry
        /programs
                /programs/Java
                        /programs/Java/java
                        /programs/Java/javac
        /etc
```

"root directory" is identified with "/" slash character
Open File Explorer and look around
FILE PATHS!

Java File

```
File students = new File("/users/phaskell/CS112/roster.txt");

boolean students.canRead();
boolean students.canWrite();
boolean students.isDirectory();
boolean students.delete(); // careful!
boolean students.exists();
int students.length(); // if 'students' is a file
String[] students.list(); // if 'students' is a directory
```

A File just links to a file in the computer filesystem. Can't do any reading/writing with it

Streams

Stream = Java object a program can read from or write to

May be attached to a

- Keyboard
- File
- Internet connection
- even a temporary chunk of memory e.g. a String or an array

In my previous job I wrote streams to talk with devices like video recorders and players

Java File Streams

Input Streams and Output Streams read and write:

- bytes and arrays of bytes: `FileInputStream, FileOutputStream`
- chars and arrays of chars: `FileReader, FileWriter`
- booleans, ints, floats, doubles, Strings, etc as readable text: `Scanner, PrintWriter`
- booleans, ints, floats, doubles, Strings, etc as raw binary data: `DataInputStream,DataOutputStream`

Other structures for streams to talk with Keyboard, internet connections, etc.

What's the difference…

Between bytes and chars?

- The "char" streams map between Character Sets (UTF-8/UTF-16) if needed

Between readable text and raw binary data?

```
double pi = 3.14159265;
PrintWriter pw = new PrintWriter("readable.txt");
DataOutputStream dos = new DataOutputStream("raw.bin");
pw.println(pi);
dos.writeDouble(pi);
```

The file **readable.txt** stores the characters '3', '.', '1', '4'… in some Character Set

The file **raw.bin** stores the 64 bits that are stored in the variable `pi`

The raw file is not readable if you look at it.
It is much smaller, much faster to read and write

```
import java.io.FileReader;
import java.io.FileWriter;

FileReader inputStream = new FileReader("input.txt");
FileWriter outputStream = new FileWriter("output.txt");

int c = inputStream.read(); // read() returns an int
while (c != -1) { // -1 not legal Unicode; signals EndOfFile
   outputStream.write(c);
   c = inputStream.read();
}
inputStream.close();
outputStream.close();
```

Note the “import” statements

Why'd we start by talking about Exceptions?

IOExceptions

Lots of File and Stream operations may raise exceptions. Most common exception type is `IOException`

- A file might not exist
- Even if the file exists, a program may not have permission to use it
- The file might not contain the kind of data we expect

An `IOException` should be handled by the program, so the user knows what is happening

```
import java.io.*;

public class TestData
{
   public static void main(String[] args) {
     try {
        PrintWriter pw = new PrintWriter(new File("test.txt"));
        for(int val = 0; val < 100; val++) {
            pw.println(val);
        }
        pw.close();
        System.out.println("Success");
      } catch (IOException e) {
        System.err.println("IOException: " + e);
      }
    }
}
```

Note the “import” wild card – matches ALL files in java.io

Type this in and run it, with good and bad file paths!

Can we Read/Write our own Classes?

`class Card, class Deck`

Yes! Must derive your class from the special built-in `class Serializable`

`Serializable` knows about all the data members in your class, and has methods that writes/reads all the values to/from a file

Use `ObjectInputStream, ObjectOutputStream`

- Also reads/writes primitive types
- Nonreadable raw-data format

A special class, like String, the Wrapper classes, etc.
Do example in Eclipse! writeObject()

Working with others…

Other computers

Programs written in other languages

Other Operating Systems

A few issues

End-of-line in text files

- Windows: \r\n
- Linux & MacOS: \n
- Many programs handle either but old ones do not
- Need to convert!

Git converts…

A few issues

End-of-line in text files

"Endianness"

- Suppose we have int x = 0x0104070a;
- When we write this to a file, do we write the 0x01 first or the 0x0a?
  - MacOS & Linux & all Java: 0x01
  - Other languages on Windows: 0x0a
  - When reading binary files written on the "other" OS, need to convert

A few issues

End-of-line in text files

"Endianness"

readUTF(), writeUTF()

- These Java methods do not just write UTF-8 values
- Hard for other programming languages to handle

Recommendations?

Audio and Video world define **byte streams**

- Not human readable, but eliminates the interoperability problems

Formats like XML only use 1-byte chars (ASCII)

- Human readable, no interop problems, slow and bloated

Only use Java?!?

One last thing

I/O to files or other devices can be SLOW

BufferedReader

- Saves data in memory before reading a big data block
- `BufferedReader br = new BufferedReader(new FileReader(…));`

"Non-synchronized" output streams

- `outStream.flush();`

Buffered means "stored up in a storage buffer"