Exception Handling

CSC 116 – Section 002

March 21, 2005

Exceptions

- "… represents an error condition that can occur during the normal course of program execution." [Wu]
- Notification of failure that terminates the normal program flow
- When an error occurs an exception is *thrown*
- You want to *catch* and handle the error properly so your program will run smoothly

Why Use Exceptions?

- Ensure that errors are handled correctly
- Example: IOException
  - Cannot open the file
  - Cannot close the file
  - Cannot read from/write to the file

Exceptions

- All Exceptions are objects
- Derived from the Exception object
- All Exception objects have two helpful methods
  - getMessage() – gets the error message
  - printStackTrace() – prints trace of the error to what line of code in your program caused the error
- You can create your own Exception objects!

Exceptions (2)

- Types of Exceptions
  - Checked: “exception that is checked at compile time” [Wu]
    - Ex: IOException and CloneNotSupportedException
  - Unchecked: “unchecked at compile time and are detected only at runtime” [Wu]
    - Ex: divide by 0 (ArithmeticException), NumberFormatException
    - However, we can check for these using try catch statements

Unchecked Exceptions

- Don’t need to check for unless you want to.
- Errors are caught and handled by system at runtime

Checked Exceptions

- If you use a method that has a checked exception type then you must handle any possible cases in which that exception is thrown
- You're code won't compile unless you handle checked exceptions
- May pass on exception with *throws* statement
- Or handle exception with *try-catch*

Throwing an Exception

- Means that your method may generate an exception but not handle it – exception propagator
- The calling method must handle the exception
- Useful when a method has no way to communicate with the user

Throwing an Exception Example

```
public void pLine () throws IOException{
   BufferedReader in = new BufferedReader(new
        FileReader("input.txt"));
   String inputLine = in.readLine();
   in.close();
   if (inputLine == null) {
        EOFException exception = new EOFException
                ("EOF when reading line");
        throw exception;
   }
   System.out.println(inputLine);
}
```

Catching an Exception

- Exception catcher
- *try* keyword used to block off code that might cause an exception
- *catch* keyword is used to hold code that is executed if an exception is thrown
- If an exception occurs then execution stops in the try block and restarts at the catch block
- One or more *catch* blocks for each *try* block
  - Most specific first
  - Most general last

Catching an Exception

- *finally* block
  - Is listed after all catch blocks
  - Runs if an error is caught or not caught
  - Used to clean up
  - Optional

Try-Catch

```
try {
    int age = Integer.parseInt(str); //could throw
}
catch (NumberFormatException e) { //exception
    System.out.println(“ERROR!”); //error code
}
catch (Exception e) { //general exception
    System.out.println(“ERROR!”); //error code
}
finally {
    //insert code here
}
```

Execution Flow

```
BufferedReader in = new BufferedReader(
        new InputStreamReader(System.in));
String line = null;
try {
        line = in.readLine();
}
catch(IOException e) {
        System.out.println(“ERROR!”);
}
finally {
        in.close();
}
System.out.println(line);
```

References

- Jason Schwarz’s Lecture 17 slides: http://courses.ncsu.edu/csc116/
- Wu – Chapter 8