Exceptions and File IO

Mikael Svahnberg*

2023-10-02

Contents

*Mikael.Svahnberg@bth.se

1 Introduction

- Barnes & Kölling Chapter 9: Well Behaved Objects
- Barnes & Kölling Chapter 14: Handling Errors
- Barnes & Kölling Chapter 14.9: File-Based input/output
- Testing for Runtime Errors
- Runtime Errors vs Compile time Errors
- Input Sanitisation
- Error Reporting vs Error Handling
- Exception Handling
- File IO

2 Testing for Runtime Errors

- Compile-time errors covered previously:
  - Syntax Errors
    - the compiler tells you what you have written wrong
    - Read the compiler errors
  - Logical Errors
    - the compiler does not know this is wrong
    - Use Unit Testing to capture this.
    - Debugger
- Runtime errors :
  - Input Testing
    - Ensure that user enters “correct” information
  - Exception handling
    - runtime exceptions
    - e.g. “file not found”, “divide by zero”, . . .

3 Handling Runtime Errors

- Handling compile-time errors is done during development
- Handling errors caught by automated tests is done during development
- Handling runtime errors *have to be* handled during runtime!

4 A Software Tester Walks into a Bar...

A Software Tester walks into a bar and orders a beer. Then they orders:

- -1 beers
- 999999999 beers
- a duck
- 0 beers
- sdlkfjhkjsdhgfk
- null

So far, so good. Then they:

- Orders 3 friends to come over for some fun.
- Unhooks the tap and orders a beer.
- Breaks all the glassware and orders a beer.
- Sets the bar on fire and orders a beer.
- Orders someone else a beer.
- Has everyone order a beer.
- Orders in russian.
- Orders a beer for later.
- Orders every beer.
- Walks into the bar backwards.
- Runs into the bar.
- Sits at the bar overnight doing nothing to see what happens.
- Tries to sell a beer.
- Quickly orders a second beer before the first is served.
- Interrupts the order midway and walks out.
- Orders a beer in IE6.
- Orders 1 ; select * from liquors; — beers.
- Orders an apostrophe and walks out without paying the bill.
- Waits for someone else to order, stands between them and the bartender, takes the drink.

... then sends them all back.

5 Detecting the Error

- Defensive Programming
  - Balance between checking everything and put reasonable trust in the program.
- Anticipating and avoiding the error
- Detecting the error state
- Deciding what to do about the error
- Deciding how much to report to the user

6 A Brief Word on Input Sanitisation

- Never ever *ever* assume that input is correct
  - User input
  - Command line parameters
  - File contents
  - Socket traffic
  - Fetched data (e.g., html, xml, json)
  - HTTP header
  - Properties/Config files
  - ...
  - `https://github.com/kuronpawel/big-list-of-naughty-strings`
- Java is safer than C++, but can still be really dangerous.
  - What do you do with the input?
    - Pass it on as a field to the database?
    - Display it in a web browser?
    - Use it as a script and interpret it?
  - What do you do if not all data is present?
  - What do you do if the data is of the wrong type?

7 Method Input: Parameters

- Are the parameters to a method within bounds?
- Are the parameters not null?
  - Are there good defaults to use instead?
- Will the method complete even with incorrect parameters?
- Will the result be meaningful?

8 Error Reporting (to the user)

- Is the error quietly fixable?
- Should the user be *informed* , or *warned*
- Should the program *exit*?
- Write to a log to facilitate debugging

```
int x = 10;
Logger l = Logger.getLogger("se.bth.example.system.Test");
l.entering("Main","main()");

l.setLevel(Level.ALL);
l.info("Current log level " + l.getLevel());

l.config("Using default configuration");
l.info("Starting...");

l.log(Level.FINE, "Still here...");
l.fine("Also still here");

l.warning("Not sure how to proceed here");

l.exiting("Main", "main()");
```

9 Error Reporting (within program)

- Magic return value that indicates error
  - `null`
  - `-1`
  - `false`
- Throw an Exception
- Set an error flag somehwere
- ...

10 Error Handling

- Check for magic error number
- Catch Exception
- Check error flags
- Deal with the error if possible
  - Maybe return some indication that there was an error

    - Log the error

- Maybe re-try the same operation
    - How many times?
    - Delay between each try?
- When all fails: re-throw exception
- **Always** clean up when you detect an error!
    - If you tried to open a file, try to close it
    - If you had an open database connection, close it
    - ...

11 Exception Handling

```
public class Document {
  private ArrayList<String> contents = new ArrayList<>();
  Document() { }

  // We handle any error (by not doing anything) and return false.
  // Or, we do what we are supposed to do and return true.
  public boolean addLine(int position, String text) {
    if ( true /* can line be added */) {
      contents.add(position, text);
      return true; // The line was added
    } else {
      return false; // The line was not added
    }
  }

  // If we throw an exception, we can use an informative class
  // so that the error can be handled in our system
  // AND we can give a good error message so that the developers
  // will know what happened.
  // There is no longer any need for a return value.
  public void removeLine(int position) {
    if (position > contents.size()) {
      throw new IndexOutOfBoundsException("Trying to remove a line outside Document bounds
    } else {
      contents.remove(position);
    }
  }

  public static void main(String [] args) {
    Document d = new Document();
    d.addLine(0, "Hello World"); // I do not need to handle the return value
    try {
      d.removeLine(99);
    } catch(IndexOutOfBoundsException e) {
      // This is the specific error message we know we might get
      System.out.println(e.getMessage());
    } catch(Exception e) {
      // Any other error message we might get. We don't want the programme
      // to terminate, so we catch, print, and move on.
      System.out.println(e.getMessage());
    } finally {
      // Cleanup regardless of whether there was an error or not
      System.out.println("Tried to remove a line");
    }

    try {
      d.removeLine(1);
      System.out.println("This line will not be executed");
    } catch(Exception e) {
    } finally {
      System.out.println("But this one will");
    }

  }
}
```

12 Checked and Unchecked Exceptions

Checked exceptions

- Can expect that the operation might fail
- Have to be caught and handled `try {} catch(SpecificException e) {}`
- May be passed on `public void doesNotHandleException() throws Exception {}`
  - Must be explicitly stated for checked exceptions
- Examples:
  - Write to disk when the disk is full
  - create a file
  - write to a network socket

Unchecked exceptions

- Should not normally happen; program error
- Are implicitly passed on
- If not caught anywhere, the program fails.
  - This is the desired behaviour; need to update program to correct.

- Examples:
  - Reading outside an array
  - Division by zero
  - Accessing a null pointer

13 Throwing and Catching Multiple Exceptions

```
public void doesNotHandleException() throws IOException, FileNotFoundException {
}

public void attemptsToHandle() {
  try {
    doesNotHandleException();
  }
  catch (IOException | FileNotFoundException e) {
    System.out.println("Error " + e.getMessage());
    e.printStackTrace();
  } catch (IllegalArgumentException e) {
    // Handle this type of exception too.
  }

}
```

14 Exceptions in C++

- More allowing than Java. Do not need to have a `Throwable` object.

```
void doSomething() {
  throw string("error");
}

void doSomethingElse() {
  throw 10;
}

int main(void) {
  try {
    //doSomething();
    doSomethingElse();
  } catch(string e) {
    cout << "caught a string " << e << endl;
  } catch(int e) {
    cout << "caught an int " << e << endl;
  }
}
```

15 File Output in Java

- Use `java.io.FileWriter`
- There are many other ways, e.g. `java.io.PrintWriter`
- Careful about the character set, åäö may trip you up.

Basic pattern:

- `try { 1. Open, 2. Write, 3. Close } catch(IOException e) {}`
- What if we fail while writing?
- What happens with close?

Preferred pattern `try-with`

- `try (resources to use ) { } catch( ... ) { }`
- Calls `close()` for you even if there is an exception.

```
import java.io.FileWriter;
import java.io.PrintWriter;

public class Outputter {
  public void basicPattern(String filename) {
    try {
      FileWriter fw = new FileWriter(filename);
      fw.write("Some text\n");
      fw.append("Some more text\n");
      fw.close();
    } catch(IOException e) {
      System.err.println("Error writing file " + filename);
      e.printStackTrace();
    }
  }

  public void preferredPattern(String filename) {
    try(FileWriter fw = new FileWriter(filename);
        BufferedWriter buf = new BufferedWriter(fw); ) {
      buf.write("Some preferred text");
      buf.newLine();
      buf.write("Some more preferred text");
    } catch (IOException e) {
      System.err.println("Error writing file " + filename);
      e.printStackTrace();
    }
  }

  public void otherWriter(String filename) {
    try ( PrintWriter out = new PrintWriter(filename)  ) {
      out.println("Some more more text");
      out.println("and yet some more");
    } catch(IOException e) {
      System.err.println("Error writing file " + filename);
      e.printStackTrace();
    }
  }


  public static void main(String [] args) {
    Outputter o = new Outputter();

    o.basicPattern("tst.txt");
    o.preferredPattern("tst2.txt");
    o.otherWriter("tst3.txt");
  }

}
```

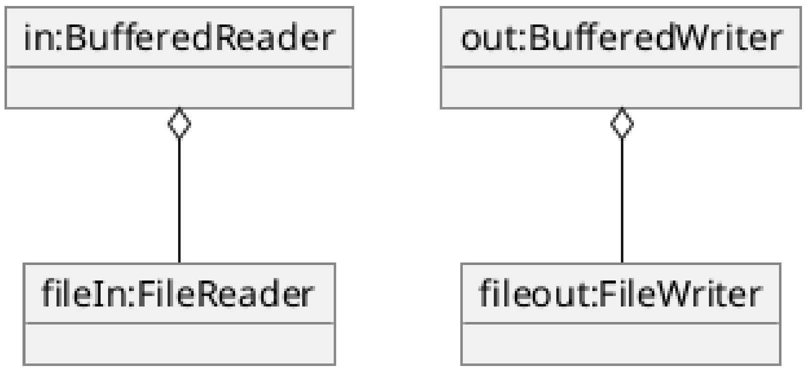
16 File Input in Java

- `java.io.FileReader` ?
  - Only has one method to read a single character.
  - `java.io.BufferedReader` to get `readLine()` method
- Two ways here too. One old-school and one newer.
- As usual, there are many other ways too.

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.nio.file.*;
import java.util.ArrayList;

public class Inputter {

  public ArrayList<String> oldSchool(String filename) {
    ArrayList<String> contents = new ArrayList<String>();
    try (
      FileReader file = new FileReader(filename);
      BufferedReader buf = new BufferedReader(file);
      ) {
      String line;
      while ((line = buf.readLine()) != null) {
        contents.add(line);
      }
    } catch (IOException e) {
      System.out.println(e);
    }
    return contents;
  }

  public ArrayList<String> hipster(String filename) {
    ArrayList<String> contents = new ArrayList<String>();
    Path path = Paths.get(filename);

    try(BufferedReader buf = Files.newBufferedReader(path)) {
      String line;
      while ((line = buf.readLine()) != null) {
        contents.add(line);
      }
    } catch (IOException e) {
      System.out.println(e);
    }

    return contents;
  }

  public static void main(String [] args) {
    Inputter in = new Inputter();
    ArrayList<String> contents;

    contents = in.oldSchool("tst.txt");
    for (String l : contents) { System.out.println(l); }

    contents = in.hipster("tst2.txt");
    for (String l : contents) { System.out.println(l); }

  }
}
```

17 Java Readers and Writers

- Conceptually, you wrap different levels of readers until you get the functionality you want
  - Typically, you will want at least a `BufferedReader` and a `BufferedWriter`



- Reader
  - BufferedReader
  - CharArrayReader
  - FilterReader
  - InputStreamReader
    - FileReader
  - PipedReader
  - StringReader
- Writer
  - BufferedWriter
  - CharArrayWriter
  - FilterWriter
  - OutputStreamWriter
    - FileWriter
  - PipedWriter
  - StringWriter
  - PrintWriter

18 File IO in C++

- Conceptually, this works the same as console input/output

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

#include <fstream>

void write(string filename) {
  ofstream out;
  out.open(filename);

  if (out.is_open()) {
    out << "You have already seen the " << flush;
    out << "output stream" << " operator in action" << endl;
    out << "some" << endl << "more" << endl << "text" << endl;
    out.close();
  }
}

vector<string>* read(string filename) {
  ifstream in(filename);
  string line;
  vector<string>* result = new vector<string>(); // Created with new since we are returnin

  if (in.is_open()) {
    while ( getline(in, line) ) { // ifstream works just like cin; we prefer std::getline(
      result->push_back(line);
    }
    in.close();
  }

  return result;
}

int main(void) {
  write("tst-cpp.txt");
  auto result = read("tst-cpp.txt"); // lazy use of auto pointer rather than specifying th

  for (auto line : *result) {
    cout << line << endl;
  }
}
```

19 Summary

- Reading from and writing to files and console
  - Also applies to values given in graphical user interface
- Sanitise inputs!
- Handling errors
  - Some operations are more likely than others to cause an error
  - Write the code such that the error is anticipated and handled
- Reporting errors with return value
- Reporting errors by throwing an exception

20 Next Lecture: More Design Patterns

- Freeman & Robson, Chapter 4: The Factory Pattern
- Freeman & Robson, Chapter 7: The Adapter and Facade Pattern
- Freeman & Robson, Chapter 9: The Iterator and Composite Pattern
- Design Principle: *Open-Closed Principle*
- Design Principle: *Depend on Abstractions*
- Design Principle: *Encapsulation*
- Design Principle: *High Cohesion*