Exceptions

Notes from the Oracle tutorial

CSCI 111
Handling an Exception
Handling an Exception

Throws exception → Method where error occurred

Forwards exception → Method without an exception handler

Catches some other exception → Method with an exception handler

main

Looking for appropriate handler

Looking for appropriate handler
Two types of Exceptions

• **Checked Exceptions:** An application is expected to recover from this event
  – Subject to a catch requirement
  – Example:
    • Providing a filename that does not exist
    • The system throws a `java.io.FileNotFoundException`
    • Your application should prompt for the correct filename
Two types of Exceptions

• **Unchecked Exceptions**: An application is not expected to recover from this event
  – Not subject to a catch requirement
  – Indicated by `RuntimeException` and its subclasses
  – Example
    • `NullPointerException`
  – **Note**: Errors are also considered unchecked exceptions
Exceptions

• Java does not require methods to catch unchecked exceptions
  – You may be tempted to only throw unchecked exceptions (i.e., `RuntimeException` or a subclass)
  – You do not have to specify catch blocks
  – You will not get compiler errors
  – Your app is not expected to recover from them
Exceptions

• Bottom line:

“If a client is expected to recover from an exception, then do not throw an unchecked exception and force the customer to handle it.”

• Code that fails to catch a checked exception will not compile.
import java.io.*;
import java.util.List;
import java.util.ArrayList;

public class ListOfNumbers {

    private List<Integer> list;
    private static final int SIZE = 10;

    public ListOfNumbers() {
        list = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            list.add(new Integer(i));
        }
    }

    public void writeList() {
        // The FileWriter constructor throws IOException, which must be caught.
        PrintWriter out = new PrintWriter("OutFile.txt");

        for (int i = 0; i < SIZE; i++) {
            // The get(int) method throws IndexOutOfBoundsException, which must be caught.
            out.println("Value at: " + i + " = " + list.get(i));
        }
        out.close();
    }
}
Catches the unchecked exception

```java
try {
    catch (IndexOutOfBoundsException e) {
        System.err.println("IndexOutOfBoundsException: " + e.getMessage());
    } catch (IOException e) {
        System.err.println("Caught IOException: " + e.getMessage());
    }
}
```

Catches checked exception thrown by FileWriter
The *finally* block

- Always executes when the try block exits
- Useful if an unexpected exception occurs
- Perfect place to perform cleanup regardless of how the try block exits (exceptions, or normally)
The *finally* block

```java
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```
Specifying exceptions thrown by a method

• It is not appropriate to catch all exceptions that can occur within a method
• It may be more appropriate to let another method further up the stack to handle the exception
• Our example:
  – If another method uses (i.e., calls) `writeList()` then let them handle the exception
  – We have to modify the signature of `writeList()`
Specifying exceptions thrown by a method

• To specify that `writeList()` can throw two exceptions (one checked and one unchecked):

```java
public void writeList() throws IOException, IndexOutOfBoundsException {
```

• Since only IOException is checked, we can also state:

```java
public void writeList() throws IOException {
```