CSSE 220

Object-Oriented Design
Files & Exceptions

Import *FilesAndExceptions* from the repo

Exam 2 – Written Part (~45 points)

- Questions about UML (~4 points)
  - For example: Here's some UML, tell what it means
  - Draw the UML from provided code
- Questions about coupling, cohesion (~5 points)
  - Based on code or UML provided on exam
- 1-2 Design Problem (include UML diagrams) (~12 points)
  - Answer include citations to Design Principles Handout
- Question about exceptions (~5 points)
- Compile-error/runtime-error/printing question (~11 points)

  – You need to be on your game for this, know 5-Steps from slides
- Tracing a recursive function (~10 points)
  - Accurate trace using diagram used for recursion tracing
  - Correct output for operation
- **Notes allowed:**
  - **You can bring 1 sheet of notes, both sides**
  - **OO Principles for 220**
  - **UML Cheat sheet**

Exam 2 – Computer Part

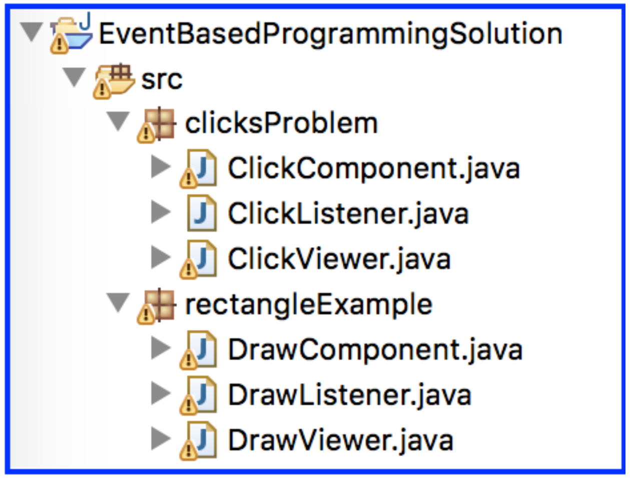
- Recursion
- Refactoring problem where you must use inheritance or interfaces to remove code duplication

Exam 2 – Take Home Part

Problem where you have to layout a GUI and handle updates using listeners

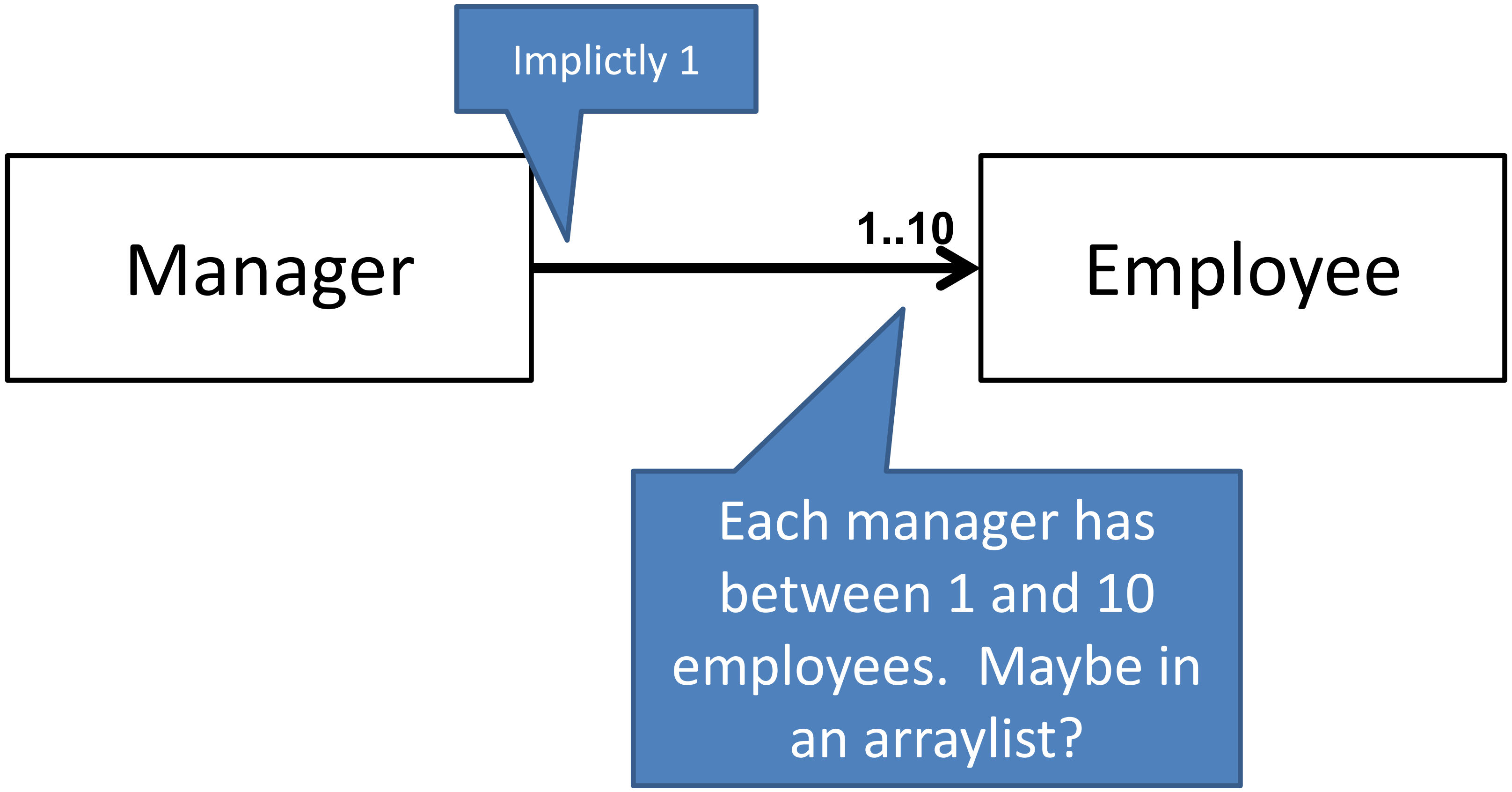
- Study *clicksProblem* and *retangleExample* from *EventBasedProgrammingSolution*

- Must have listeners for Buttons
- Must have listeners for Mouse – mouse moves, mouse clicks, etc.

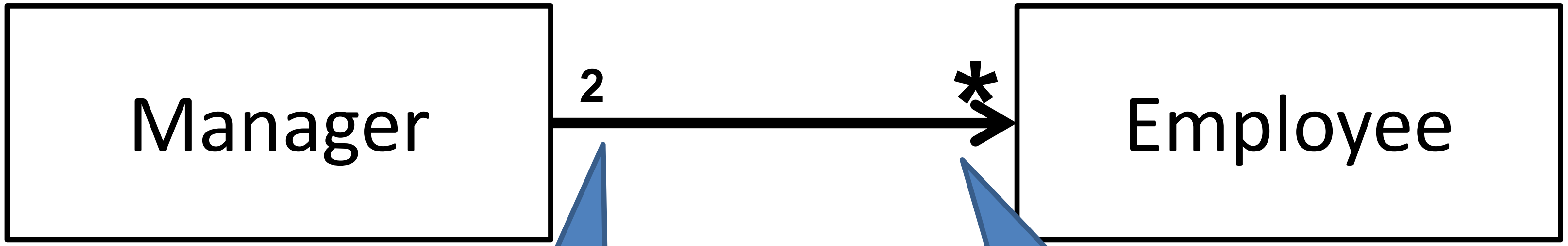


```
EventBasedProgrammingSolution
  src
    clicksProblem
      ClickComponent.java
      ClickListener.java
      ClickViewer.java
    rectangleExample
      DrawComponent.java
      DrawListener.java
      DrawViewer.java
```

More UML Notation: Cardinality


Implictly 1
Manager
1..10
Employee
Each manager has between 1 and 10 employees. Maybe in an arraylist?

More Cardinality

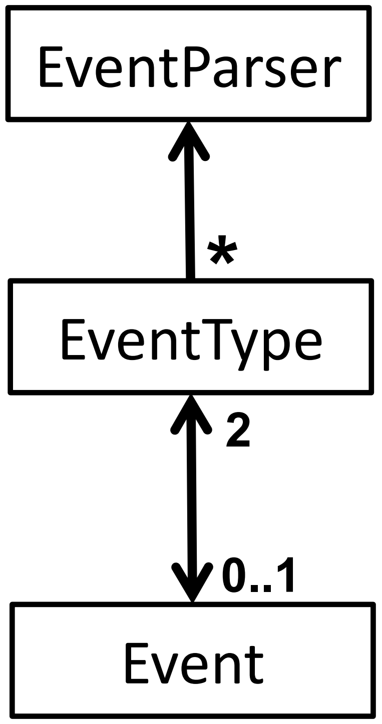


Every employee has exactly 2 managers. Note that this can be used even if there is no reference from Employee to Manager

Managers have any number of employees.

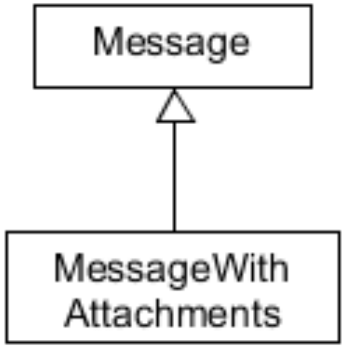
The * means “zero to infinity” – any arbitrary number. You can also occasionally see something like 4..* to mean 4 or more.

What does this diagram mean?

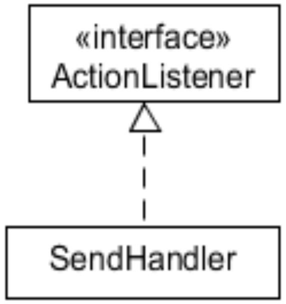


Summary of UML Class Diagram Arrows

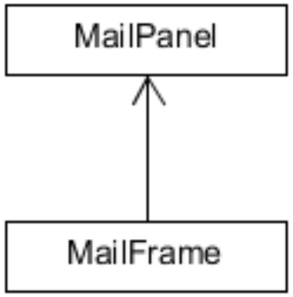
Inheritance (is-a)



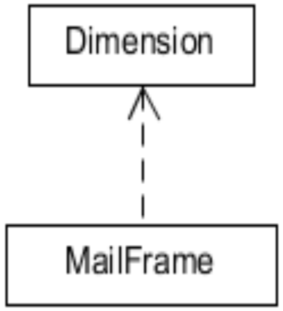
Interface Implementation (is-a)

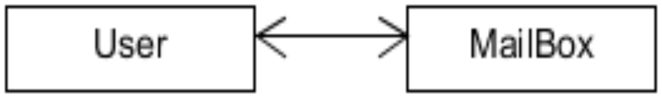


Association (has-a-field)

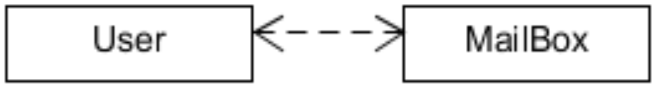


Dependency (depends-on)

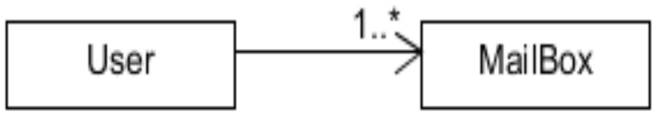




Two-way Association



Two-Way Dependency



Cardinality
(one-to-one, one-to-many)
One-to-many is shown on left

Q1

Answer Quiz Question #1

On Exam 2 – Written Part

- Lots of UML Diagrams
- Don’t rely on UML Cheat sheat – will take too much time

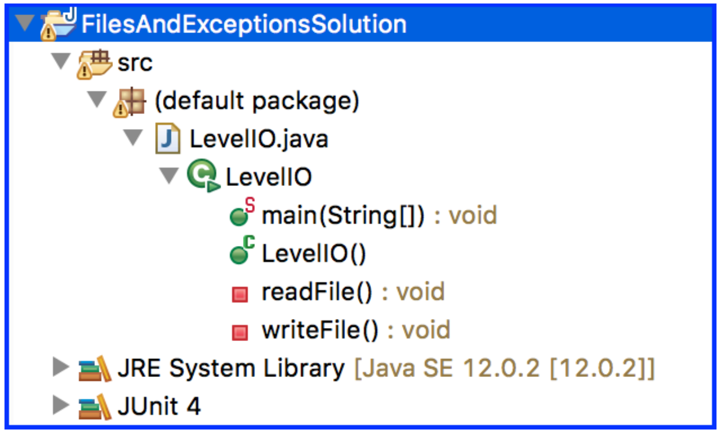
Reading & writing files
When the unexpected happens

FILES AND EXCEPTIONS

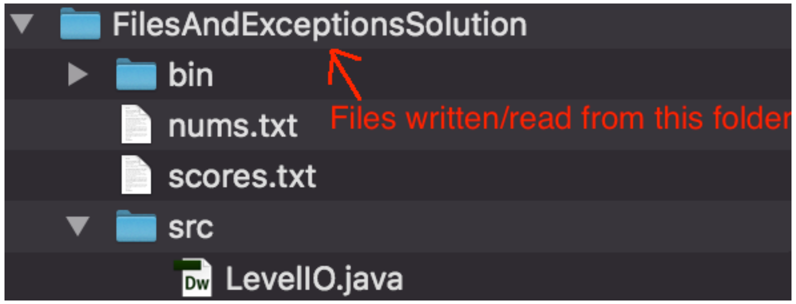
File I/O: Key Pieces

- Input: `File` and `Scanner`
- Output: `PrintWriter` and `println`
- ☺ Be kind to your OS: `close()` all files
- Letting users choose: `JFileChooser` and `File`
- Expect the unexpected: `Exception` handling
- Refer to examples when you need to…

Live code/modify a LevelIO



FilesAndExceptionsSolution
- src
  - (default package)
    - LevelIO.java
      - LevelIO
        - main(String[]) : void
        - LevelIO()
        - readFile() : void
        - writeFile() : void
- JRE System Library [Java SE 12.0.2 [12.0.2]]
- JUnit 4



FilesAndExceptionsSolution
- bin
- nums.txt
- scores.txt
- src
  - LevelIO.java

Files written/read from this folder

Demonstrate:

1. How to have Eclipse auto generate try-catch
2. How to have Eclipse auto add throw to header
3. How PrintWriter() throws exception on locked folder
4. Note where files are read/written relative to .java files
5. How File() throws exception on no file found
6. How to print additional File info from File object

Exception – One Approach

- Write operation so it returns false when operation detects some kind of error

- Two minutes to think this through and consider why this might not work in all situations

Exception – One Approach

- Write operation so it returns false when operation detects some kind of error

- Function might already be returning a value, e.g., an int
- Caller may not write code to examine returned value
- No information other than called operation failed

Exception – What, When, Why, How?

- What:
  - Used to signal that something in the code has gone wrong
- When:
  - An error has occurred that cannot be handled in the current code
- Why:
  - Breaks the execution flow and passes exception up the stack – It is like hitting the *ejection button*, i.e., normal *return* not executed

Exception – How?

- Throwing an exception:

throw new EOFException("Missing column");

The call to **throw** "hits the ejection button"
The call to **new** creates an exception object
This new exception object is "thrown" back to caller

- Either your code *throws* an exception when it detects some kind of problem
- Or some existing Java operation your code has called throws an exception when it detects some kind of exception

Q5

Exception – How?

- Handling (catching) an exception:

```
try {
    // code that contains a call to some operation
    // that could throw an exception
}
catch (ExceptionType ex) {
    //code to handle exception
} // end try-catch
```

When exception is caught you can:

1. Recover from the error OR
2. exit gracefully

- #1 above is often not possible, when writing the code you don't know what to do to fix problem
- #2 is more likely, e.g., log an error into a error log file, then re-throw exception, which will usually cause program to abort

Q5

What happens: A few examples

On next few slides are a number of different scenarios

1) What happens: no exception is thrown?

```
Scanner inScanner;
try {
        inScanner =
                new Scanner(new File(“test.txt”));
        //code for reading lines
} catch (IOException ex) {
        JOptionPane.
                showMessageDialog("File not found.");
} finally {
        inScanner.close();
}
```

If this line is successful

Code continues on

The catch never executes

This runs after code in try completes

2) What happens: exception is thrown?

```
Scanner inScanner;
try {
        inScanner =
```

If this line throws exception

```
                new Scanner(new File(“test.txt”);
        //code for reading lines
```

Code after exception never executes

```
} catch (IOException ex) {
        JOptionPane.
```

This is the next line executed

```
                showMessageDialog("File not found.");
} finally {
        inScanner.close();
```

After catch is executed, this runs

```
}
```

3) When exception is not handled?

```
public String readData(String filename)
                          throws IOException {
        Scanner inScanner =
                new Scanner(new File(filename));
        //code for reading lines
        inScanner.close();
}
```

If this line throws exception

Code does not execute,
Method breaks immediately

main -> readAllFiles -> readData

If unhandled, exception propagates to method that called it, then up the call stack all the way up to main

A Checkered Past

- Java has two sorts of exceptions

1. **Checked exceptions**
2. **Unchecked exceptions**

A Checkered Past

- Java has two sorts of exceptions

1. **Checked exceptions**: compiler **checks** that calling code is not ignoring the problem
   - Used for **expected** problems, e.g., file not found

What to do:

A. Catch it, and ignore it by doing nothing

B. Catch it, and re-throw a different exception that is not checked:

```
throw new RuntimeException(" … ")
```

C. Add "throw" clause to header of your operations

A Checkered Past

- Java has two sorts of exceptions

2. **Unchecked exceptions**: compiler lets us ignore these if we want
   - Used for fatal or avoidable problems
   - Are subclasses of RunTimeException or Error

A Tale of Two Choices

Dealing with checked exceptions

1. Can **propagate** the exception
   - Just declare that our method will pass any exceptions along…
   - `public void loadGameState() throws IOException`

   - Used when our code isn't able to rectify the problem

2. Can handle the exception
   - Used when our code can rectify the problem

Q6

A Tale of Two Choices

Dealing with checked exceptions

1. Can **propagate** the exception
   – Just declare that our method will pass any exceptions along…

```
public void loadGameState() throws IOException
```

Our method called loadGameState:

- Calls some Java operation that throws IOException
- It does not attempt to try-catch handle IOException
- So we add “throws IOException” to our method header

Q6

A Tale of Two Choices

Dealing with checked exceptions

2. Can handle the exception
   - Used when our code can rectify the problem
   - In this case we add a try-catch to our operation's implementation

Q6

Handling Exceptions

- Use try-catch statement:

```
try {
    // potentially "exceptional" code
} catch (ExceptionType var) {
    // handle exception
}
```

Can repeat this part for as many different exception types as you need.

Related, try-finally for clean up:

```
try {
    // code that requires "clean up"
} // then maybe some catches
finally {
    // runs even if exception occurred
}
```

Q7

Exception Activity – 15 Minutes

- Do both of these activities:

1. Look at the code in FileAverage, focusing on the use of exceptions

2. Solve the problems in FileBestScore

Arcade Game - Cycle 0

Cycle 0 – UML diagram
- Will morph over time – that's not a problem
- The idea is to get a start at an overall design
- If try to do this HW w/o doing an initial design,
  you will run into a lot of dead ends, and waste lots of time

- Joe - Go to Word doc – skim through
- Afterward have team members introduce themselves