Exception Handling

- General idea
- Checked vs. unchecked exceptions
- Semantics of . . .
  - throws
  - try-catch
- Example from text: DataAnalyzer
  - running it
  - modifying it

Announcements

- Lab this week is based on textbook example we'll discuss today.
- See Announcements for office hours schedule this week.

Lecture references

- Big Java
- **Core Java**, Volume I, 8th Ed, by Horstmann and Cornell, Prentice Hall, 2008

Errors and recovery options

- Suppose an error occurs while our program is running.
- Good program responses:
  - Report the error and…
  - return to safe state and allow user execute other commands
  - or allow user to save all work and and terminate program gracefully [From Core Java]
- Not so good program responses:
  - program crashes
  - program is mum about the error

Exceptions idea

- Allows us to separate point of error *detection* from point of error *recovery*

- First, an example to refer to in our discussions…

Textbook example

- Code is in Section 11.5
- Problem:
  - *read in a bunch of data from a file with a specific format and process the data (computes the sum)*
- Example file **`in1`** (first line is number of values):

```
3
1.45
-2.1
0.05
```

Why exceptions? I

- Recall: Exceptions allow us to separate point of error *detection* from point of error *recovery*
- Why?
- cleaner code for normal case. E.g.:

```
for (int i = 0; i< numVals; i++) {
   readValue(in, i);
}
```

Why exceptions? II

- May not have enough info in method where error is detected
- Ex: suppose if we get a bad data value in **`readValue`**, we want to ask for a new file name.
- But the code that gets file name is in **`main`**
- Call stack:

`main` calls

error can be handled here

`readFile` calls

`readData` calls

`readValue` calls

`in.nextDouble()` - ERROR detected here

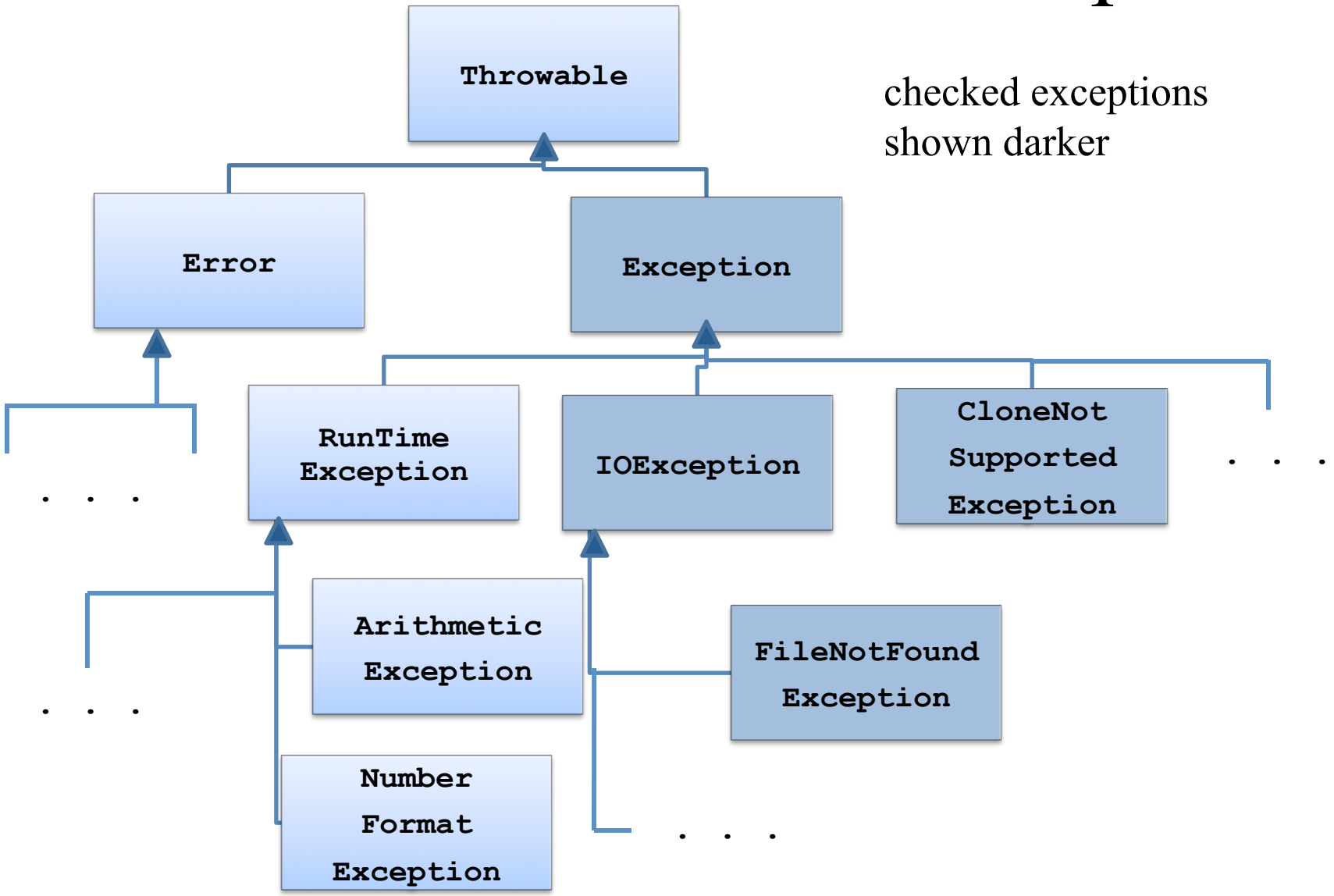
Why exceptions? III

- All that catch-throw stuff looks confusing…
- Is there some alternative?
  - returning an error code from our function
- but…
- in Java we can only return one value.
- use return value for the normal value,
- throw an exception for one or more abnormal situations

Why exceptions? IV

- Some Java library methods throw exceptions,
- So, even if we don't want to use exceptions
- we need to know a little bit about the mechanism to *even compile* any of our code that uses those parts.
- Our first such example: trying to open a file that isn't there.

Classification of Java Exceptions



Unchecked Exceptions

- Unchecked: **`Error`**, **`RunTimeException`**
  - **`RunTimeException`**: Don't throw, don't catch:

    Fix the bug in your program!
    - e.g., ArrayIndexOutOfBoundsException, NullPointerException, ArithmeticException
  - **`Error`**: Not your fault; internal error you can't recover from
    - e.g., ran out of heap memory

Checked Exceptions

- Checked: all other **`Exception`** subclasses
  - These are largely user errors that you may handle with the exception mechanism.
  - e.g., EOFException, FileNotFoundException,
  - These are the ones that your code has to do something about (Why IV earlier)

Java exception handling mechanism

- **`throw`** / **`throws`**: someone else will deal with the exception
- **`try`** - **`catch`**: my code will deal with the exception
- **`finally`**: a way to release resources before exiting (via throw or return)

Java method throws an exception. What do I do?

- Easiest response: we throw the exception onwards.
- Ex: (**`Total.java`**, Sect.11.1)

```
public static void main(String[] args) throws
                                    FileNotFoundException {
    . . .
    File inputFile = new File(inputFileName);
    Scanner in = new Scanner(inputFile);
    // ... code to read the file
    . . .
}
```

if file not found:
main exits immediately; program crashes

(Not in **`main`**) Java method throws an exception. What do I do? (cont.)

- Easiest response: we throw the exception to our caller.
- If you're not main, and you don't know how to handle exception, perfectly ok to throw it to caller:

```
public void read(String fileName) throws
                                  FileNotFoundException {
    . . .
    File inputFile = new File(fileName);
    Scanner in = new Scanner(inputFile);
    . . .
}
```

file not found: control returns to caller immediately. caller has to catch or throw

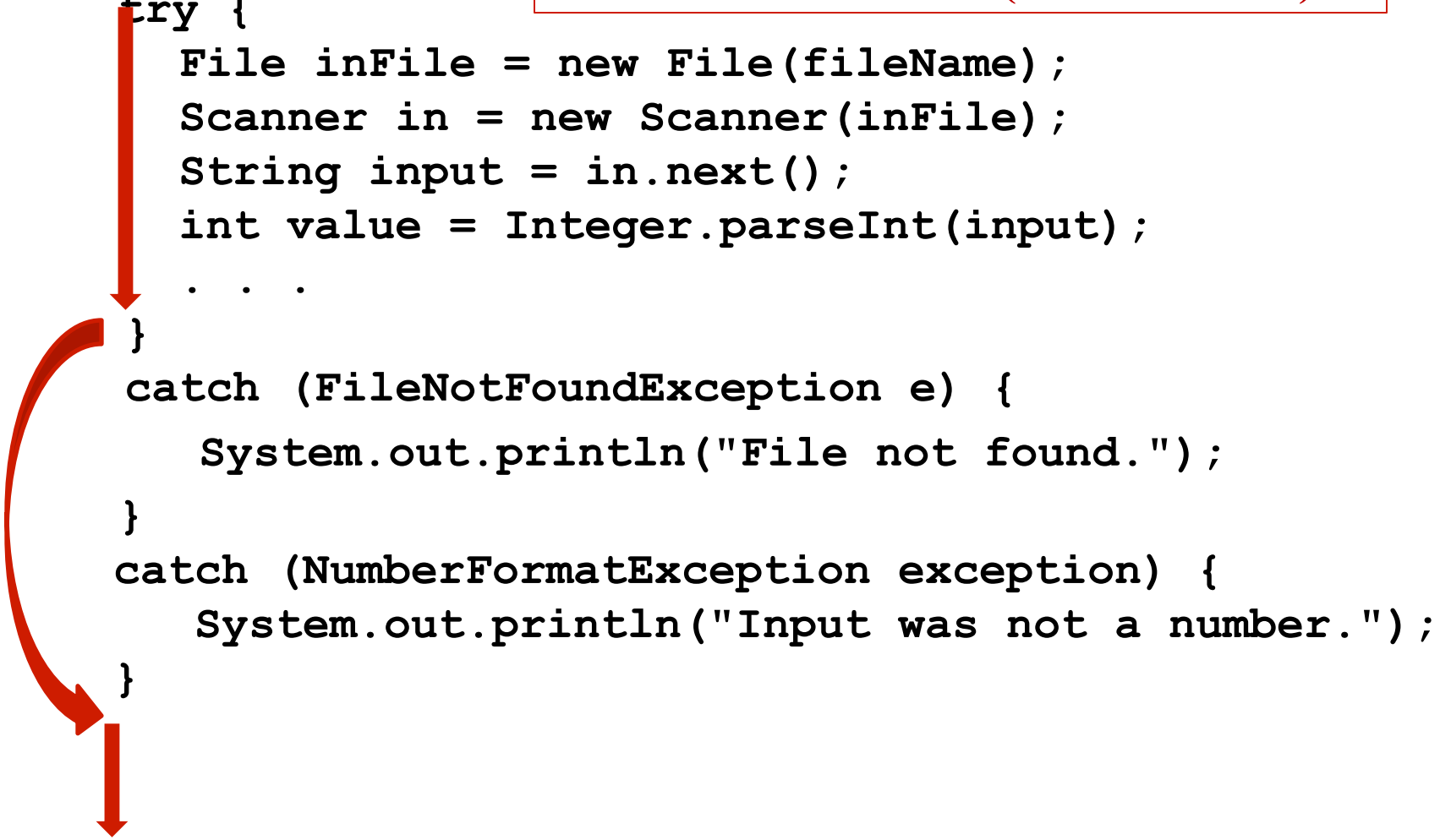
Catching Exceptions

- To catch an exception…
  - have to put the code that may throw the exception in a **`try`** block of a **`try-catch`** statement.
  - the part that handles the exception is in the **`catch`** block

```
try {
   // some code that may throw an exc.
}
catch (ExceptionType e) {
   // some code that handles the exception
   // (e.g., reports the error and recovers,
   //  or reports and exits)
}
```

Ex: Try-catch flow-of-control

Normal flow of control (no exc. thrown)



```
try {
   File inFile = new File(fileName);
   Scanner in = new Scanner(inFile);
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (FileNotFoundException e) {
   System.out.println("File not found.");
}
catch (NumberFormatException exception) {
   System.out.println("Input was not a number.");
}
```

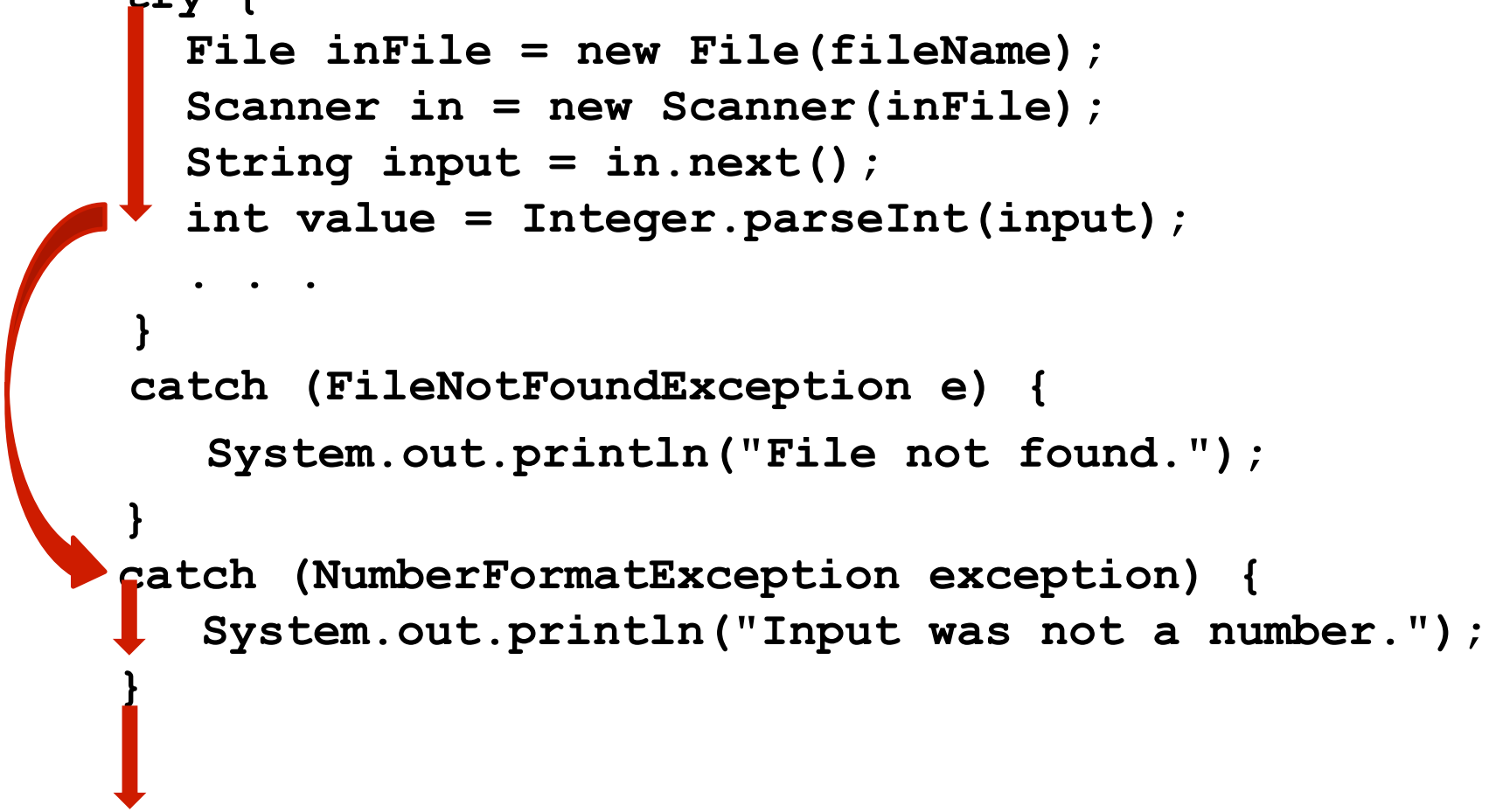
Ex: Try-catch flow-of-control

File not found exception

```
try {
   File inFile = new File(fileName);
   Scanner in = new Scanner(inFile);
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (FileNotFoundException e) {
   System.out.println("File not found.");
}
catch (NumberFormatException exception) {
   System.out.println("Input was not a number.");
}
```

Ex: Try-catch flow-of-control

NumberFormatException thrown by parseInt



```
try {
   File inFile = new File(fileName);
   Scanner in = new Scanner(inFile);
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (FileNotFoundException e) {
   System.out.println("File not found.");
}
catch (NumberFormatException exception) {
   System.out.println("Input was not a number.");
}
```

Ex: Try-catch flow-of-control

NumberFormatException thrown by parseInt; no handler for it

don't need "throws" in header because unchecked exception

```
public void someMethod( . . . ) {
   . . .
   try {
     File inFile = new File(fileName);
     Scanner in = new Scanner(inFile);
     String input = in.next();
     int value = Integer.parseInt(input);
     . . .
   }
   catch (FileNotFoundException e) {
      System.out.println("File not found.");
   }
   . . .
}
```

exits method immediately

Ex: What if exception matches multiple catch clauses

FileNotFoundException is a subclass of IOException; FileNotFound thrown by Scanner

```
try {
   File inFile = new File(fileName);
   Scanner in = new Scanner(inFile);
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (FileNotFoundException e) {
   System.out.println("File not found.");
}
catch (IOException exception) {
   exception.printStackTrace();
}
```

matches most specific Exc. type

only one catch clause executed

What about recovery?

- So far have only seen printing a message.
- Recovery is in the context of a larger program
- We'll do a larger example presently.
- But first…

Do not squelch exceptions!

- Suppose my code can throw a checked exception:

```
File inFile = new File(fileName);
Scanner in = new Scanner(inFile);
```

- Bummer, it won't compile.
- this will shut it up!

```
try {
   File inFile = new File(fileName);
   Scanner in = new Scanner(inFile);
   . . .
}
catch (FileNotFoundException e) { }
```

Don't do this!

- Means that instead of handling the error, when that error comes up the behavior is undefined.

Case Study

- Code is in Section 11.5 Handling Input Errors
- Problem:
  - *read in a bunch of data from a file with a specific format and process the data (computes the sum)*
- Example file **`in1`** (first line is number of values):

```
3
1.45
-2.1
0.05
```