

# Exception Handling

- General idea
- Checked vs. unchecked exceptions
  - unchecked example: checking preconditions
- Semantics of . . .
  - throws
  - try-catch
- Example from text: DataAnalyzer

# Announcements

- Lab this week is based on the textbook example we'll discuss today.
- Reminder: Wellness Day Friday (no office hours or lab meeting)
- Reminder: Daylight savings time starts this Saturday night (local time changes to PDT)
- Unofficial HW (will discuss in Tues. lect):  
Write code to use Java **`Arrays.sort`** method to sort an array of Java **`Rectangle`** objects in increasing order by area.

# Lecture references

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

# Exception handling

- Mechanism to help make bullet-proof programs:
- Desirable for our apps to be able to deal with any kind of external error. Examples:
  - bad user input
  - bad file formats
  - missing files
  - failed database connection

# 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
- Not so good program responses:
  - program crashes
  - program is mum about the error

# Exception handling 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
```

# Possible user problems with data for DataAnalyzer example

Example file **in1** (first line is number of values):

```
3  
1.45  
-2.1  
0.05
```



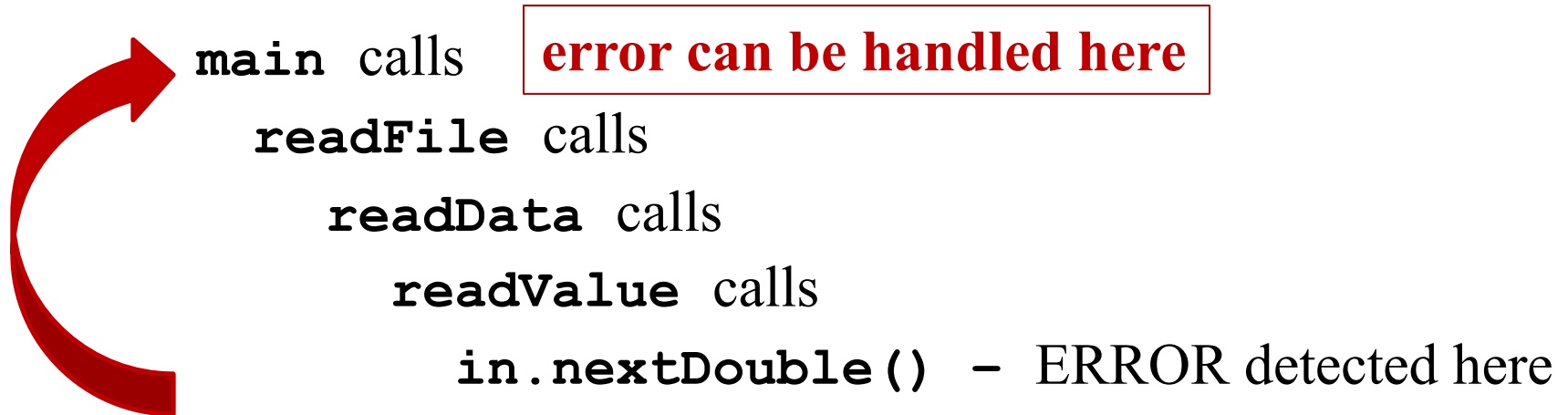
# Why exceptions? I

- 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:



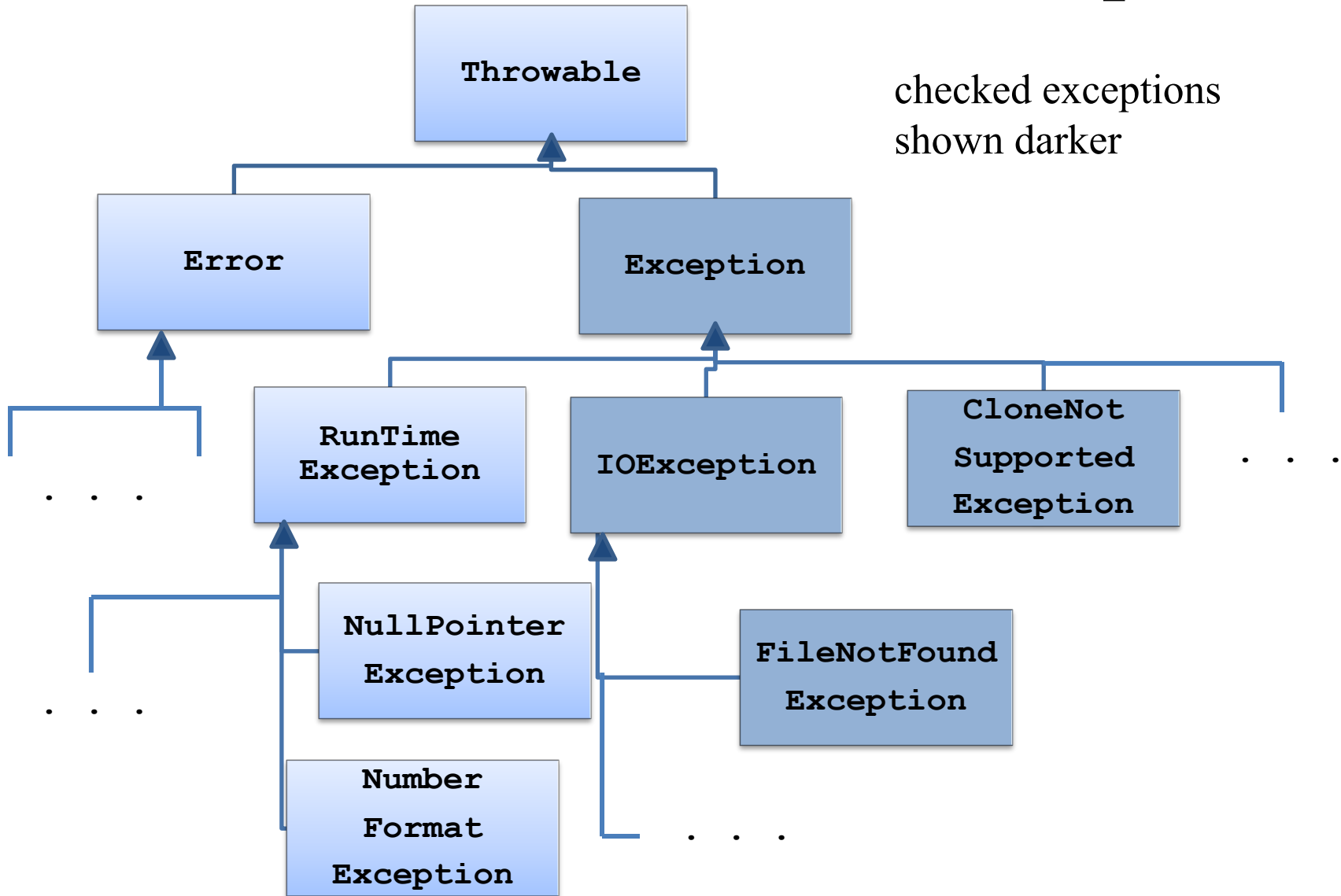
# Why exceptions? III

- 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.
  - for constructor: not even one return 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

# Using unchecked exceptions to check method preconditions

```
// Withdraws money from the bank account.
// @param amount the amount to withdraw
// @throws IllegalArgumentException
//         if amount <= 0 or exceeds the balance.
public void withdraw(double amount) {
    if (amount > balance) {
        throw new IllegalArgumentException("withdrawal of "
            + amount + " exceeds balance of " + balance);
    }
    if (amount <= 0) {
        throw new IllegalArgumentException(
            "amount must be > 0");
    }
    balance = balance - amount;
}
```

# 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 *must* do something about (Why IV earlier)



# Unchecked IO-related exceptions

- Major category of *checked* exceptions:  
**IOException**
- However, **next()** and **nextInt()** can throw *unchecked* exceptions
  - (**InputMismatchException** and **NoSuchElementException**)
- Can handle issue without using exception handling.

# Java exception handling mechanism

- **throw / throws**: someone else will deal with the exception
- **try - catch**: my code will deal with the exception
- **try-with-resources**: automatically closes a file upon completion whether an exception happened or not

# 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 int[] 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.");  
}
```

The diagram illustrates the flow of control in a try-catch block. A red arrow points from the 'try' keyword to the first catch block. Another red arrow points from the first catch block to the second catch block. A final red arrow points from the end of the second catch block, indicating the flow after the try-catch structure.

# 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.");  
}
```


The diagram illustrates the flow of control in a try-catch block. A red arrow points from the start of the try block to the first catch block (FileNotFoundException). Another red arrow points from the end of the first catch block to the second catch block (NumberFormatException). A third red arrow points from the end of the second catch block to the end of the try-catch structure.



# Ex: Try-catch flow-of-control

**NumberFormatException thrown by parseInt;  
no handler for it**

```
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.");  
    }  
    . . .  
}
```




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

**exits method immediately**

# Ex: What if exception matches multiple catch clauses

**FileNotFoundException is a subclass of IOException; FileNotFoundException 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();  
}
```

The diagram illustrates the exception matching process. A red arrow points from the `try` block to the first `catch` clause (`FileNotFoundException`). A second red arrow points from the first `catch` clause to the second `catch` clause (`IOException`). A final red arrow points from the second `catch` clause down, indicating the flow of execution.

**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: DataAnalyzer

- 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
```

# DataAnalyzer: where to see the code

- Code is linked on web on week-by-week page:  
open each file in a different tab.

(Also available on Vocareum in Lecture code area under 03-10)

- Poll links for asynchronous participation:
  - [File not found poll](#)
  - [in.badFloat poll](#)