# CS18000: Problem Solving and Object-Oriented Programming

## File I/O and Exception Handling

(revised 11/24/23)

# Video 1
# Basics of File I/O

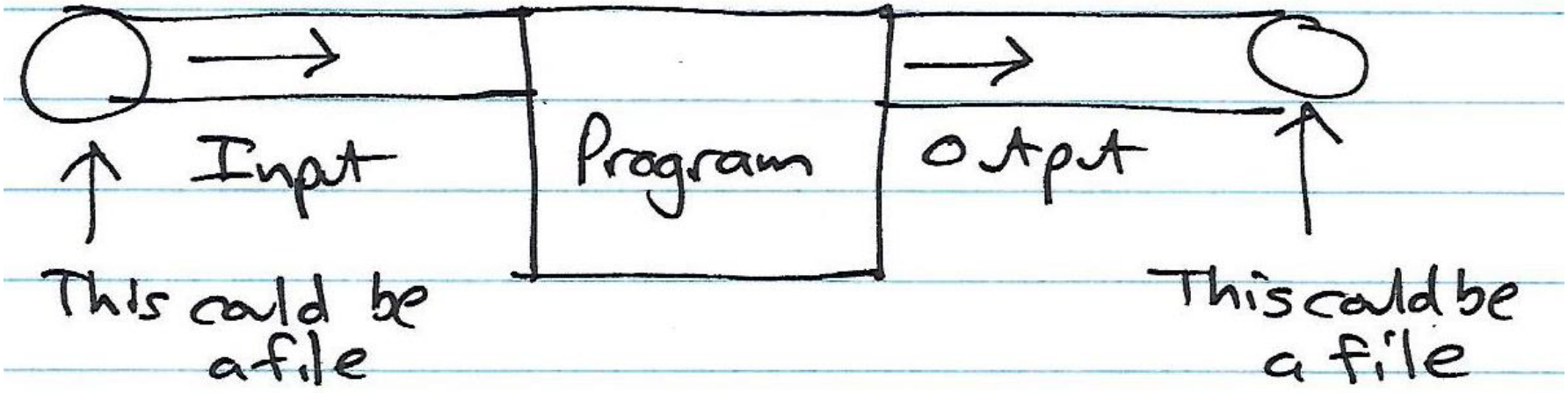
# External Communication

File I/O

# Persistence of File Storage

- RAM comes and goes
  - Programs crash
  - Systems reboot
- Files last (well … comparatively speaking…)
- Programs save data to files to
  - recover from program crashes and system reboots
  - provide as input to other programs
- File I/O operations extend naturally to communication between programs

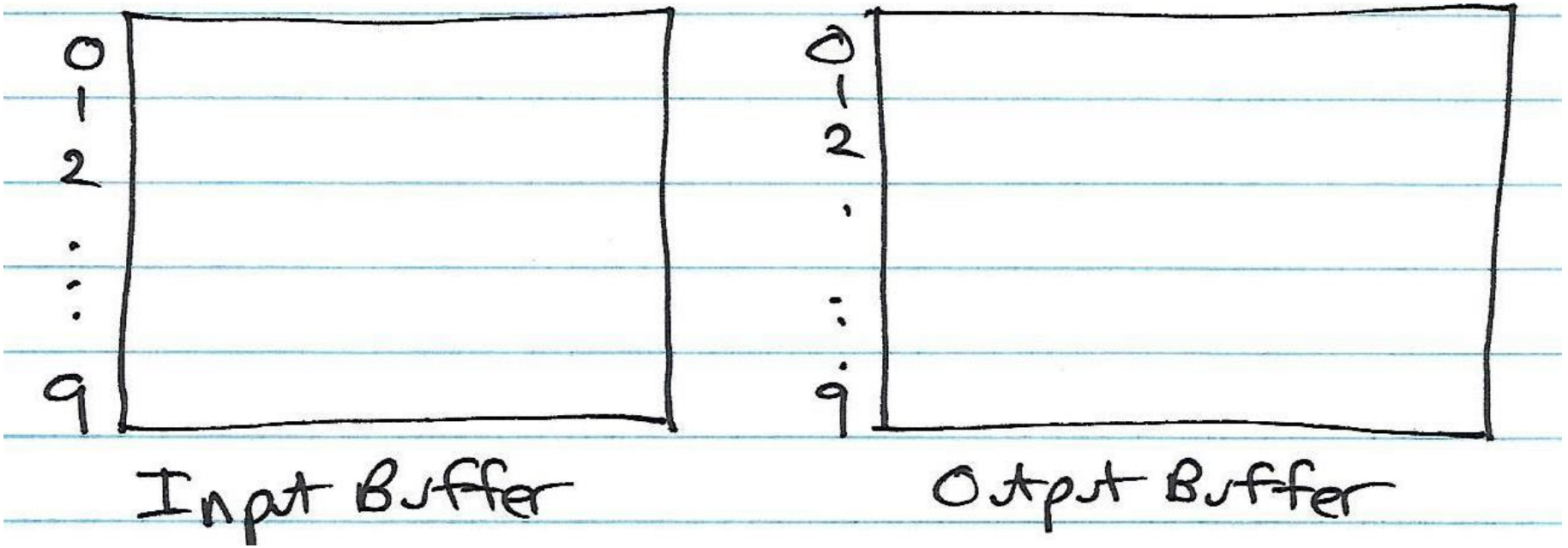
# Input and Output “Pipes”



# Files and Java

- Java is (or tries to be) platform independent
- Provides abstractions for files and file systems
- File class
  - But, file name is operating system (OS) dependent
  - And, file directory conventions are OS-dependent (e.g., path name of user home directory)
  - So, there are limits to OS independence
- Three layers of abstraction in Java for file I/O
- Ultimately, all data stored as a stream of bytes

# The Implementation of Buffering



# The Importance of Buffering

- Without buffering, each read or write may generate physical disk access
- Can be extremely slow for large volumes of data
- Buffering has OS create internal array
  - OS reads “more than needed” on input, keeps rest for next call to read method
  - OS doesn’t send output “right away” to disk drive, waits a while in case another write comes along
  - Important to close file (or flush buffers) when done

# Generic File Operations (1)

- Open:
  - Files must be opened before they can be used
  - Open method indicates “for reading”, “for writing”, or “both”
  - May also indicate “append” mode
  - Allows operating system to establish “buffers” and other state information about the file being read or written
- Read
  - Transfers data from the file (or input stream) to the user process
  - Specific method signatures indicate the type of data being transferred (byte, int, String, Tree, etc.)
- Write
  - Transfers data from the user process to the file (or output stream)
  - Specific method signatures indicate the type of data being transferred (byte, int, String, Tree, etc.)

# Generic File Operations (2)

- File position
  - Sets the “current input position” to a specific byte address in the file
  - Can be used to skip over data in the file; or back up to read data again
  - Can be used to “rewind” the file to start reading from the beginning again
- Close
  - Ensures that any “queued data” is “flushed” from the operating system buffers
  - Frees any operating system resources being dedicated to managing the file

# Video 2
# Low-Level, High-Level, and Object I/O

# File I/O Layers in Java

- Low-Level
  - "Raw" data transfer: byte-oriented
  - Classes: FileOutputStream, FileInputStream
- High-Level
  - Java primitive types
  - Classes: DataOutputStream, DataInputStream
- Object I/O
  - Java object types
  - Classes: ObjectOutputStream, ObjectInputStream

*Ultimately, all data stored as a sequence of bytes*

# Example: Low-Level I/O

```
import java.io.*;

public class LowLevelIO {
    public static void main(String[] args) throws IOException {
        File f = new File("lowlevel");

        FileOutputStream fos = new FileOutputStream(f);
        fos.write(42);
        fos.close();

        FileInputStream fis = new FileInputStream(f);
        int i = fis.read();
        System.out.printf("Read %d\n", i);
        fis.close();
    }
}
```

# Example: High-Level I/O

```
import java.io.*;

public class HighLevelIO {
    public static void main(String[] args) throws IOException {
        File f = new File("highlevel");

        FileOutputStream fos = new FileOutputStream(f);
        DataOutputStream dos = new DataOutputStream(fos);
        dos.writeInt(1000);
        dos.close();

        FileInputStream fis = new FileInputStream(f);
        DataInputStream dis = new DataInputStream(fis);
        int i = dis.readInt();
        System.out.printf("Read %d\n", i);
        dis.close();
    }
}
```

dos builds on fos

dis builds on fis

# Tricky Bits

- You must keep track of what you're doing!
- Data values must be read in the same order in which they were written
  - write int, long, long, boolean, double, float, char
  - read int, long, long, boolean, double, float, char
- If you try to read an int, but a double is next in the stream, you'll get garbage

# Example: (1)

```
import java.io.*;

public class ObjectIO {
    public static void main(String[] args) throws Exception {
        File f = new File("object");

        FileOutputStream fos = new FileOutputStream(f);
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        Tree tree1 = new Tree(42, "elm");
        oos.writeObject(tree1);                 // write the object out
        oos.close();

        FileInputStream fis = new FileInputStream(f);
        ObjectInputStream ois = new ObjectInputStream(fis);

        Tree tree2 = (Tree) ois.readObject();   // read the object back

        ois.close();

        System.out.printf("tree1 = %s\n", tree1);
        System.out.printf("tree2 = %s\n", tree2);
    }
}
```

oos builds on fos

ois builds on fis

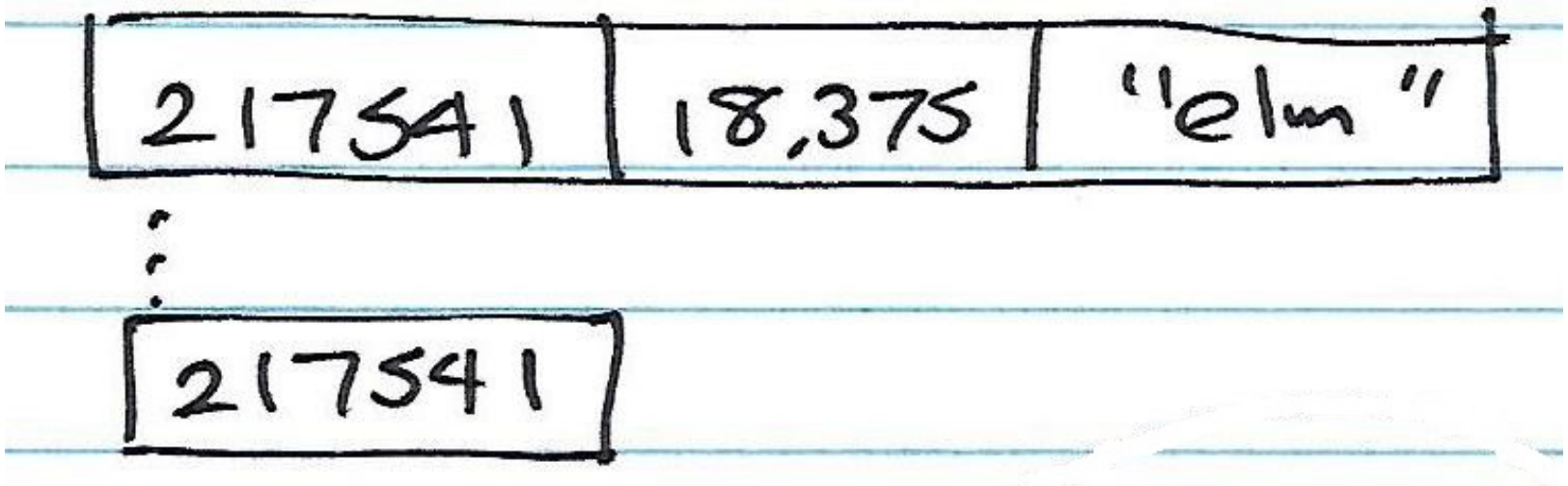
# Example: Object I/O (2)

```
class Tree implements Serializable {
     double circumference;
     String species;

     Tree(double circumference, String species) {
          this.circumference = circumference;
          this.species = species;
     }

     public String toString() {
          return String.format("%x: circumference = %d,  species = %s",
                    hashCode(), circumference, species);
     }
}
```

# Serializable



# Video 3
# Text I/O

# File Content Types

- Can consider file contents in two categories
- Text (e.g., *.java, *.txt)
  - Store human-readable, character data
  - Mostly platform independent (except EOL)
- Binary (e.g., *.class, *.exe)
  - Not (generally) human readable
  - Store any kind of data
  - Requires specific programs to “make sense” of it

# Writing and Reading Text

- Java handles translation from internal primitive format to human-readable text
- Writing
  - Class: PrintWriter (favored, more platform independent)
  - Class: PrintStream for System.out (but out of favor)
- Reading
  - Classes: FileReader and BufferedReader
  - Also, Scanner
- Note: BufferedReader is more efficient than Scanner (only important for high volumes of I/O)

# Example: TextIO (1)

```
import java.io.*;

public class TextIO {

    public static void main(String[] args) throws IOException {
        File f = new File("textio.txt");

        // open FileOutputStream in append mode (true)
        FileOutputStream fos = new FileOutputStream(f, true);

        // use PrintWriter--similar to PrintStream (like System.out)...
        PrintWriter pw = new PrintWriter(fos);
        pw.println("our old friend");
        pw.close();

// continued...
```

# Example: TextIO (2)

```
// ... continued

        // read what we just wrote...
        FileReader fr = new FileReader(f);
        BufferedReader bfr = new BufferedReader(fr);
        while (true) {
            String s = bfr.readLine();
            if (s == null)
                break;
            System.out.println(s);
        }
        bfr.close();
    }
}
```

# Video 1
# Introduction to Exceptions

# Exceptions

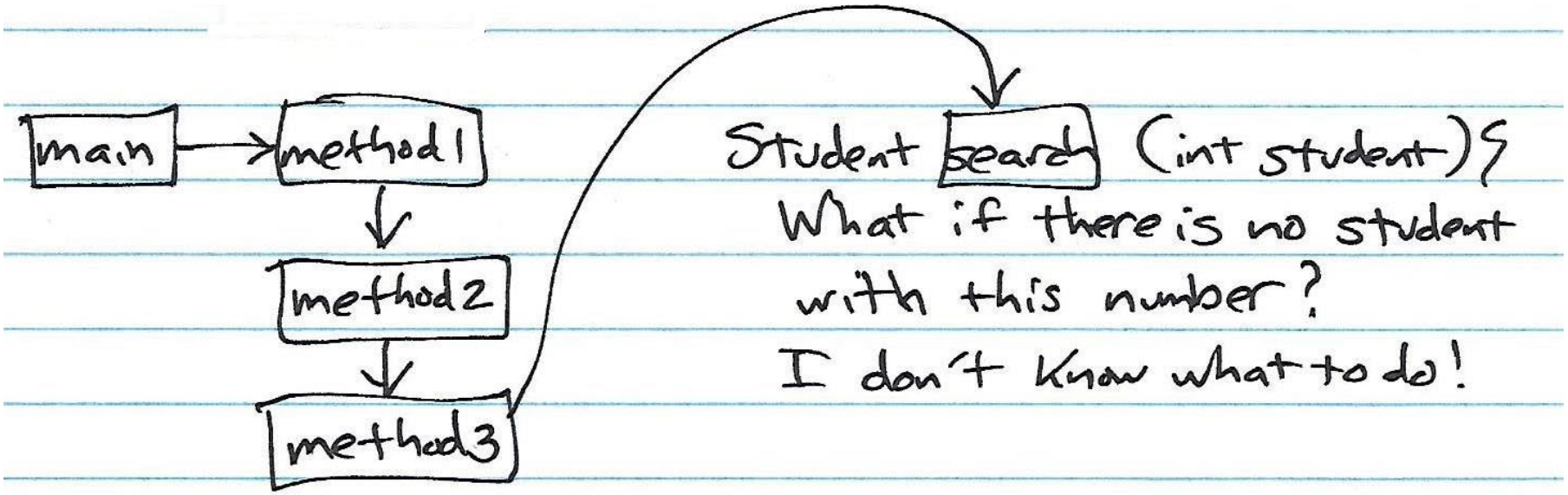
try-catch

throw

# Handling Error Situations

```
public class Summer {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int number;  // number that is input
        int sum = 0; // sum of values
        int c = 0; // how many values read
        double average; // average value
        while (in.hasNextInt()) {
            number = in.nextInt();
            c = c + 1;
            sum = sum + number;
        }
        if (c > 0) {
            average = sum / c;
            System.out.printf("%d values, sum %d, average %f", c, sum, average);
        } else
            System.out.printf(“no values, no sum, no average");
    }
}
```

# What to do when an error occurs?



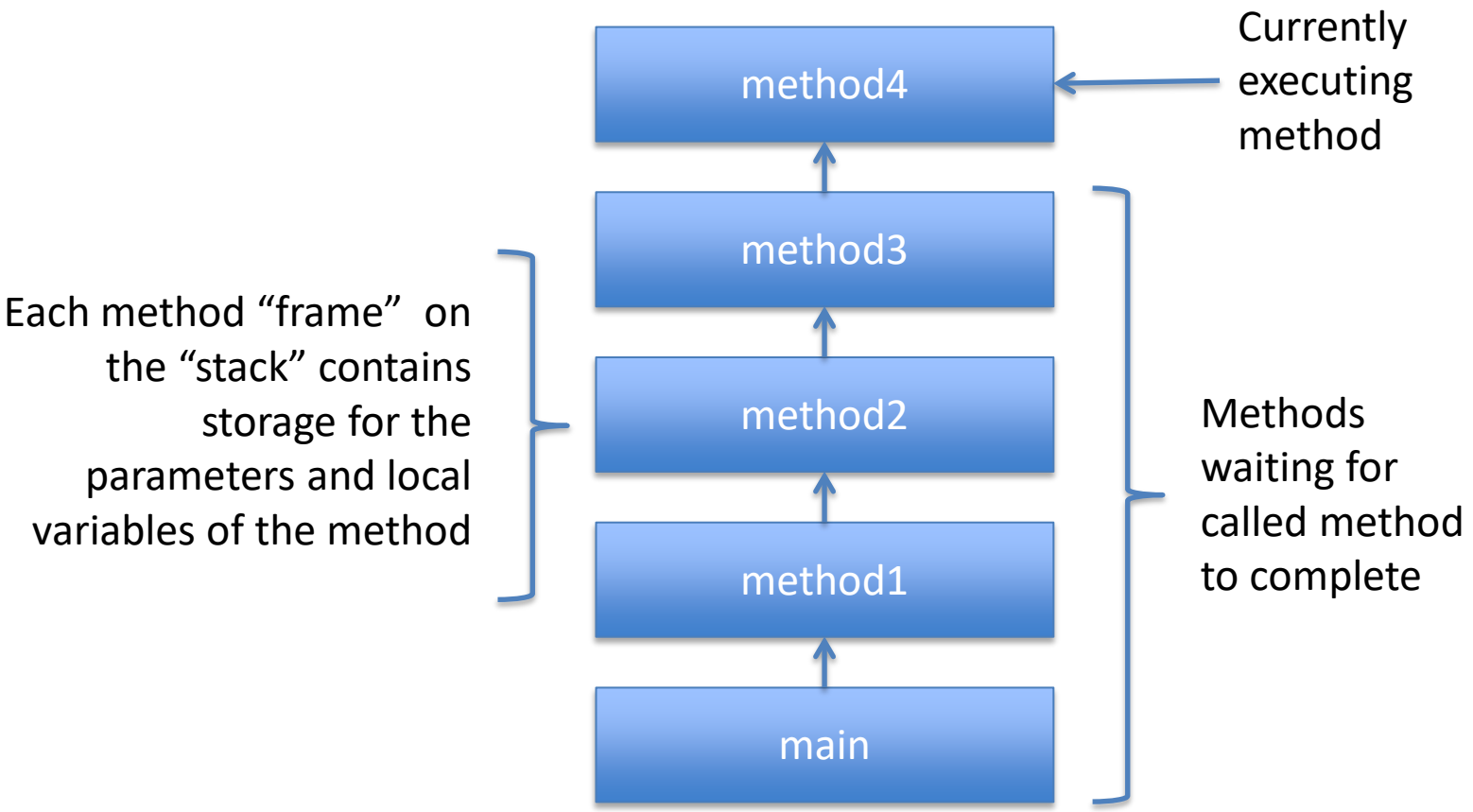
# What to do when an error occurs?

- Old style: return an “error code”
- Caller must check on each call
  - Did the method return an error?
  - Requires a special value to indicate error
- Example:
  - indexOf() method used to retrieve index position at which a particular character appears in a string
  - If specified character is not found, indexOf() returns -1
  - Programmer must check for -1

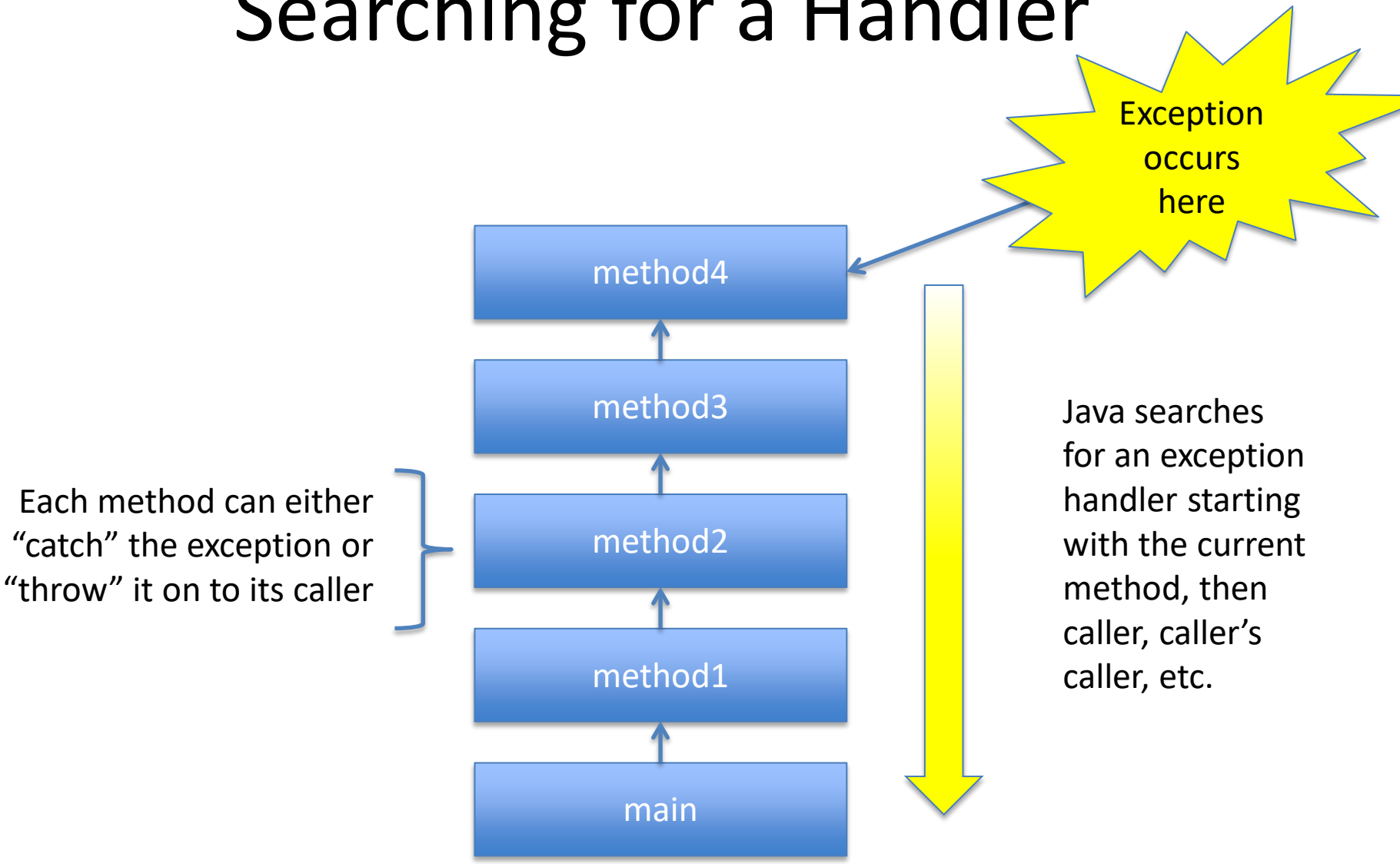
# Java Approach: Exceptions

- Write code without worrying about checking for errors
- When an error is detected, an exception is "thrown"…
  - Java system stops execution of the current method
  - Searches for an "exception handler" to deal with the problem
- Search begins in the current method and continues to
  - caller -> caller's caller -> caller's caller's caller ->
  - …-> main -> …

# The Call Stack



# Searching for a Handler

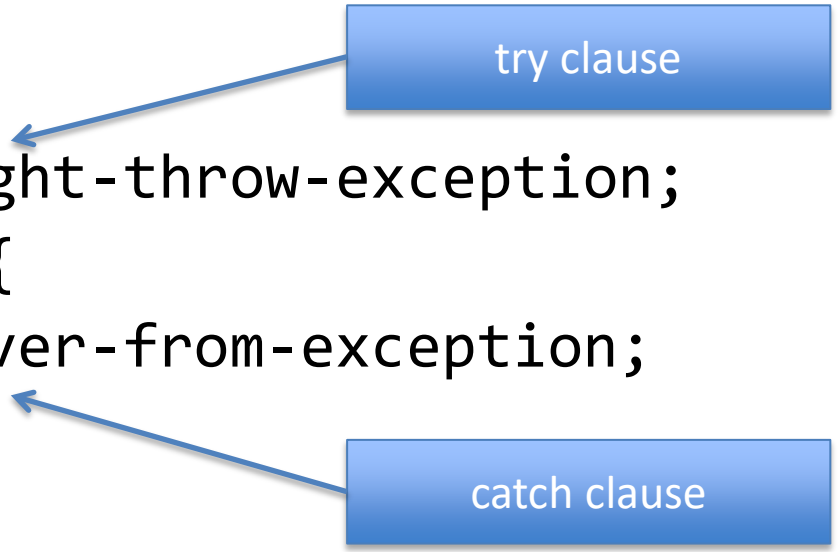


# Video 2
# The Exception Class

# Catching an Exception: Basic Syntax

Basic syntax of the try-catch statement…

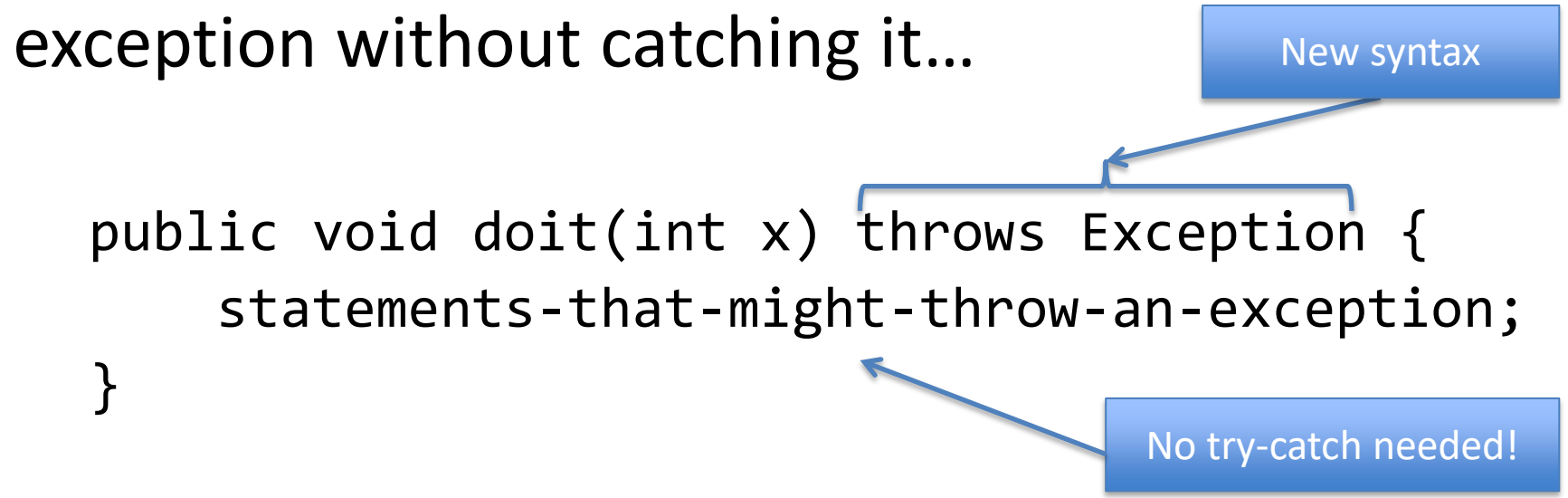
```
try {
    statements-that-might-throw-exception;
} catch (Exception e) {
    statements-to-recover-from-exception;
}
```




Note: “Exception” is a class name, not a reserved word; e is an object reference.

# Passing the Buck: Throws

A method can declare that it throws an exception without catching it…

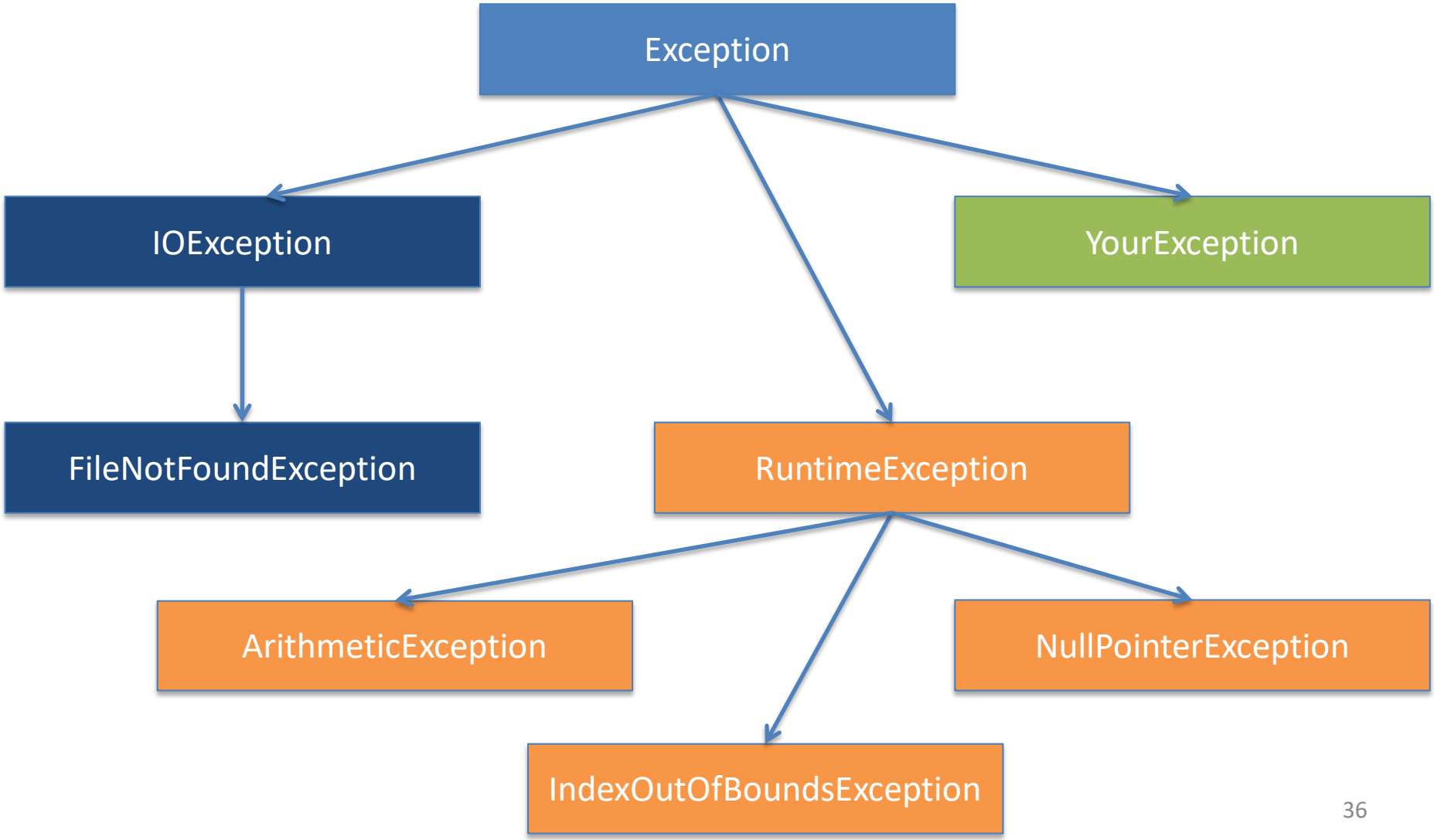



Note: “throws” is a keyword, “Exception” is a class name

# Exception Class

- Exceptions are objects
- The exception object is an instance of
  - class `Exception`, or
  - a subclass of `Exception`
- Created using `new` (just like any object)
- Two useful methods…
  - e.getMessage() get the associated text message
  - e.printStackTrace() prints the current call stack

# Exception Class Hierarchy



# Checked vs. Unchecked Exceptions

- The RuntimeException class and its subclasses are “unchecked” exceptions:
  - Generally indicate program or JVM error (null pointer, arithmetic, invalid array index, etc.)
  - Typically: no recovery is possible; program crashes
- All other Exceptions are “checked”
  - Generally indicate “user” error (e.g., file not found)
  - Must check for them (try-catch or throws)
  - Typically: recoverable (e.g., prompt user again)

# EOF: Unchecked Exception

```
import java.util.Scanner;

public class EOF {
    public static void main(String[] args) {
        FileReader fr = new FileReader(f);
        Scanner s = new Scanner(fr);

        while (true) {
            String word = s.next();
            System.out.println(word);
        }
    }
}
```

***Throws NoSuchElementException at end of file***

# EOF: Catching NoSuchElement

```
import java.util.Scanner;
import java.util.NoSuchElementException;

public class EOF {
    public static void main(String[] args) {
        FileReader fr = new FileReader(f);
        Scanner s = new Scanner(fr);
        while (true) {
            try {
                String word = s.next();
                System.out.println(word);
            } catch (NoSuchElementException e) {
                System.out.printf("NoSuchElementException: %s\n”,e.getMessage());
                break;
            }
        }
    }
}
```

# Scanner: Catching FileNotFound

```
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;

public class LineCounter {
    public static void main(String[] args) {
        File f = new File(args[0]);

        try {
            FileReader fr = new FileReader(f);
            Scanner s = new Scanner(fr);
            int c = 0;
            while (s.hasNextLine()) {
                s.nextLine();
                c++;
            }

            System.out.printf("read %d lines from file %s\n", c, f);
        } catch (FileNotFoundException e) {
            System.out.printf("Exception: %s\n", e.getMessage());
        }
    }
}
```

# Video 3
# Advanced Exception Handling

# Making Your Own Exception Class

```
public class StudentNotFoundException extends Exception {
    public StudentNotFoundException (String message) {
        super (message);
    }
}

public class FindStudent {
    public Student search (int student) throws
                StudentNotFoundException {
    if (...) {
        throw new StudentNotFoundException
            (Integer.toString(student));
        }
    }
}
```

# Typical Exception Handling Situation

```
try {
    …
    method1(…);
    …
} catch (StudentNotFoundException e) {
    statements-to-recover;
}
```

# Catching Multiple Exceptions

- It is possible to catch multiple exceptions from one `try`
- Catches must be ordered from lowest subclass to highest superclass

```
try {
    … statements-that-may-throw-exceptions;
} catch (StudentNotFoundException e) {
    // code to handle student not found
} catch (NullPointerException e) {
    // code to handle null pointer
} catch (Exception e) {
    // code to handle all other exceptions
}
```

# Finally Clause

- Finally, if present, a "`finally`" clause is executed after all other try/catch statements
- The `finally` clause is guaranteed to execute, even if earlier clause returns

```
try {
    … statements-that-may-throw-exceptions;
} catch (StudentNotFoundException e) {
    // code to handle student not found
…
} finally {
    // code to clean things up
}
```

# try-with-resources statement

- Instead of a finally block to ensure that a resource is closed you can use a try-with-resources statement

- A resource is an object that must be closed after the program is finished with it

# try-with-resources statement

```
static String readFirstLineFromFile(String path) throws
IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path)))
   {
        return br.readLine();
   }
    catch (StudentNotFoundException e) {
    // code to handle other exceptions
   }
}
```

# try-with-resources statement

- Resource declared in the try-with-resources statement is a `BufferedReader`
- `BufferedReader br` must be closed after the program is finished with it
- `BufferedReader br` will be closed regardless of whether the try statement completes normally or abruptly