

# Programming Languages and Techniques (CIS1200)

## Lecture 31

### I/O & Histogram Demo

#### Chapter 28

# Announcements (1)

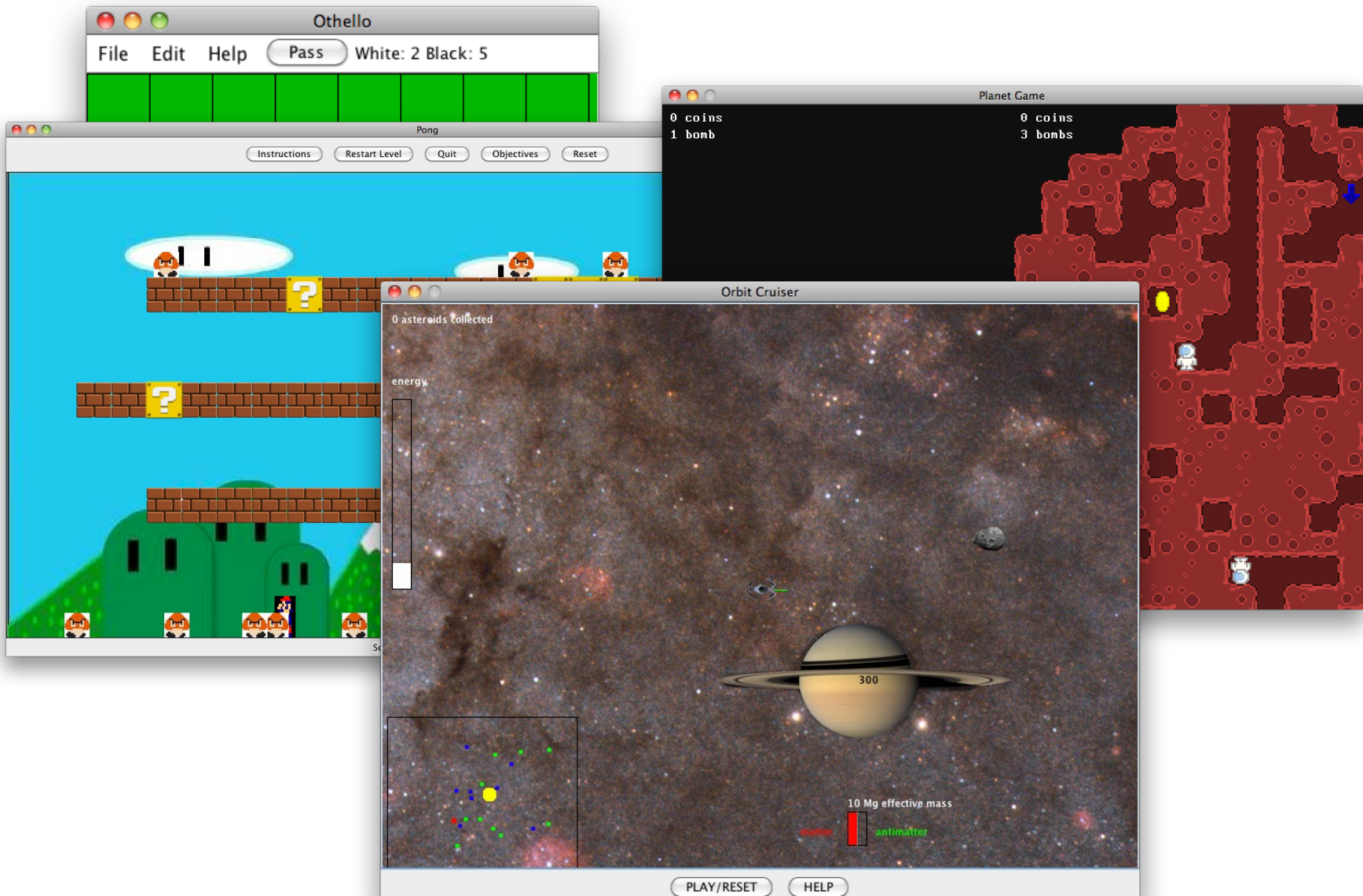
- Midterm 2: Friday, November 15<sup>th</sup>
  - Similar to Midterm 1
  - Content: HW 4 – 6, Chapters 11-21 (Java Arrays) and Chapter 32 (Encapsulation) of lecture notes
- Dr. Sheth will have extra office hours
  - Thursday from 2-4pm, Levine 264
- HW08: TwitterBot\*
  - Released soon; due on November 26<sup>th</sup>
  - Practice with I/O and Collections

\* Maybe should be called "XBot" or "TheProjectFormerlyKnownAsTwitterBot"?

# Announcements (2)

- Midterm 2: Friday, November 15
  - Coverage: up to Monday, Oct. 28 (Chapters 11-21, 32)
  - During lecture (001 @ 10.15am, 002 @ noon)  
Last names: A – Z                      Meyerson Hall B1
  - 60 minutes; closed book, closed notes
  - Review Material
    - old exams on the web site (“schedule” tab)
  - Review Session
    - **Tonight**, Towne 100 (will be recorded)

# HW9: Game Project



# HW9: Game project

- Game Design Proposal Milestone Due: (8 points)  
**Thursday, November 21<sup>st</sup> at Midnight = 11:59PM!**
  - (Should take about 1 hour)
  - Submit on GRADESCOPE
  - TAs will give you feedback soon
- Final Program Due: (92 points)  
**Monday, December 9<sup>th</sup> at 11:59pm**
  - Submit zipfile online, submission *only* checks if your code compiles
  - IntelliJ is **strongly recommended** for this project
  - You may distribute your game (after the deadline) if you do not use any of our code
- Grade based on demo with your TA during/after reading days
  - Grading rubric on the assignment website
  - Recommendation: don't be too ambitious.
- ***NO LATE SUBMISSIONS PERMITTED***

# Announcements (4)

- TA position applications are available
  - CIS 1100, 1200, 1600, 1210 (see <https://tinyurl.com/2tn2t22f>)
  - Other CIS and NETS classes (see <https://www.cis.upenn.edu/ta-information/>)
  - Accepting applications until Friday, November 22<sup>nd</sup>
  - Intro CIS TA Panel
    - Recording should be available

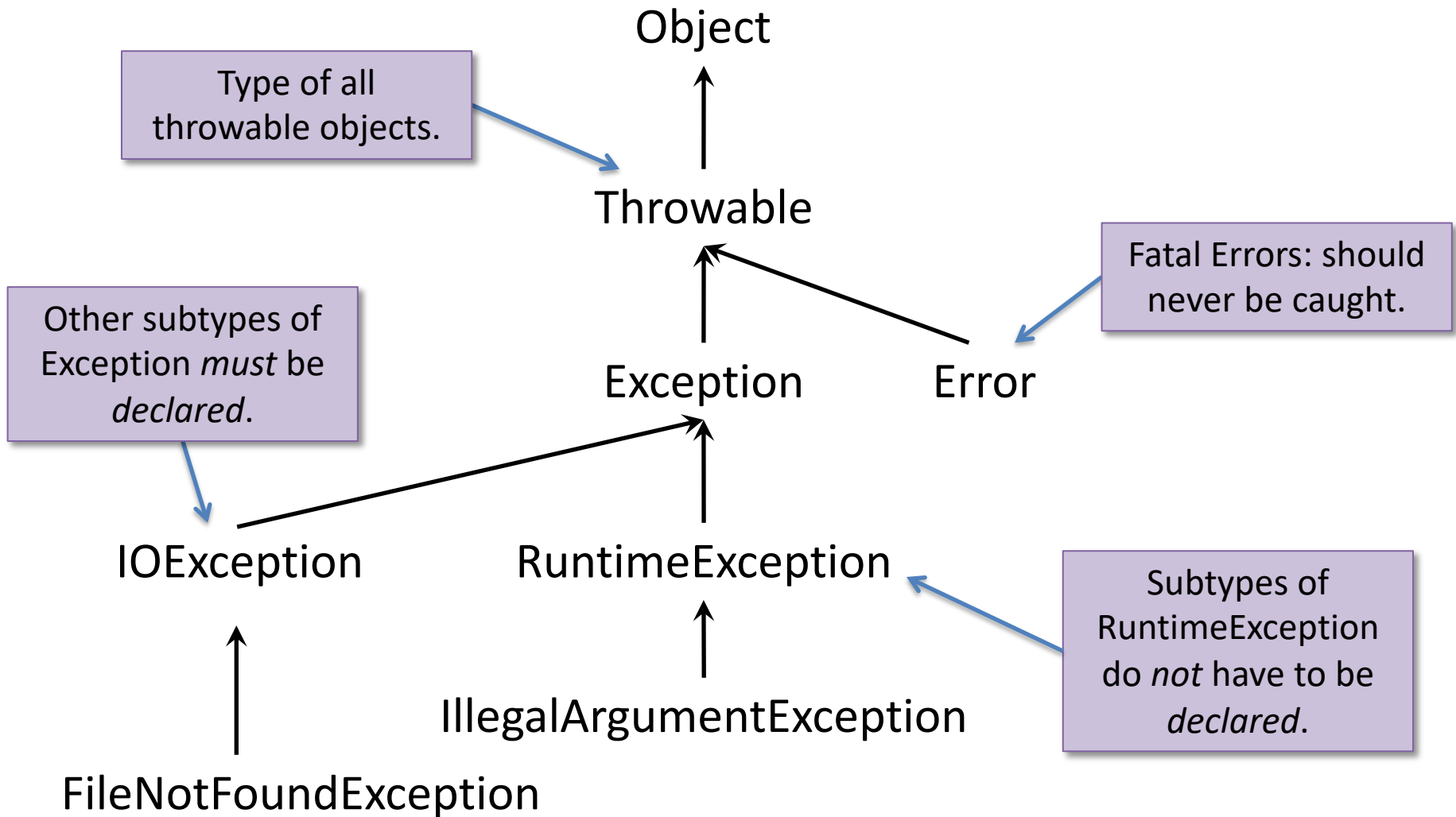
# Review: Exceptions

# Exceptions

- Exceptions are just objects that affect control flow:
- Raise an exception with:  
**throw new** ExceptionType();
  - aborts the current execution context (workspace)
  - "unwinds" the stack, searching for a matching catch block
- Handle exceptions using try/catch:  
**try** { /\* code \*/ }  
**catch** (ExceptionType e) { /\* handler \*/ }
  - runs code
  - if code raises an exception that is a subtype of ExceptionType, intercept its stack unwinding and run the handler



# Exception Class Hierarchy



# Checked (Declared) Exceptions

- Exceptions that are subtypes of `Exception` but not `RuntimeException` are called *checked* or *declared*.
- A method that might throw a checked exception must declare it using a “throws” clause in the method type.
- The method might raise a checked exception either by:
  - directly throwing such an exception

```
public void maybeDoIt (String file) throws Exception{  
    if (...) throw new Exception(); //directly throw  
    ...
```

- or by calling another method that might itself throw a checked exception

```
public void doSomeIO (String file) throws IOException {  
    Reader r = new FileReader(file); // might throw  
    ...
```

# Unchecked (Undeclared) Exceptions

- Subclasses of RuntimeException *do not* need to be declared via “throws”
  - even if the method does not explicitly handle them.
- Many “pervasive” types of errors cause RuntimeExceptions
  - NullPointerException
  - IndexOutOfBoundsException
  - IllegalArgumentException

```
public void mightFail (String file) {  
    if (file.equals("dictionary.txt")) {  
        // file could be null!  
        ...  
    }  
}
```

- The original intent was that such exceptions represent disastrous conditions from which it was impossible to sensibly recover...

# Declared vs. Undeclared?

- Tradeoffs in the software design process:
- *Declared*: better documentation
  - forces callers to acknowledge that the exception exists
- *Undeclared*: fewer static guarantees (compiler can help less)
  - but, much easier to refactor code
- In practice: test-driven development encourages “fail early/fail often” model of code design and lots of code refactoring, so “undeclared” exceptions are prevalent.
- A reasonable compromise:
  - Use declared exceptions for libraries, where the documentation and usage enforcement are critical
  - Use undeclared exceptions in client code to facilitate more flexible development

# Finally

```
try {  
    ...  
} catch (Exn1 e1) {  
    ...  
} catch (Exn2 e2) {  
    ...  
} finally {  
    ...  
}
```

- A `finally` clause of a `try/catch/finally` statement *always* gets run, regardless of whether there is no exception, a propagated exception, or a caught exception.

# Using Finally

```
class C {  
    public void foo() {  
        this.bar();  
        System.out.println("here in foo");  
    }  
    public void bar() {  
        try {  
            this.baz();  
        } catch (Exception e) {  
            System.out.println("caught");  
        } finally { System.out.println("finally"); }  
        System.out.println("here in bar");  
    }  
    public void baz() {  
        throw new RuntimeException();  
    }  
}
```

What happens if we do `(new C()).foo()` ?

Answer: 4

1. Program prints only "finally"
2. Program prints "here in bar", then "here in foo", then "finally"
3. Program prints "finally", then "caught", then "here in foo"
4. Program prints "caught", then "finally", then "here in bar", then "here in foo"

# Good Style for Exceptions

- In Java, exceptions should be used to capture *exceptional circumstances*
  - Try/catch/throw incur performance costs and complicate reasoning about the program, don't use them when better solutions exist
- *Re-use existing exception types* when they are meaningful to the situation
  - e.g. use NoSuchElementException when implementing a container
- Define your own subclasses of Exception if doing so can convey useful information to possible callers that can handle the exception.

# Good Style for Exceptions

- It is often sensible to catch one exception and re-throw a different (more meaningful) kind of exception.
  - *e.g.*, when implementing `WordScanner` (in upcoming lectures), we catch `IOException` and throw `NoSuchElementException` in the `next` method.
- Catch exceptions as near to the source of failure as makes sense
  - *i.e.*, where you have the information to deal with the exception
- Catch exceptions with as much precision as you can
  - BAD:**     `try {...} catch (Exception e) {...}`
  - BETTER:** `try {...} catch (IOException e) {...}`

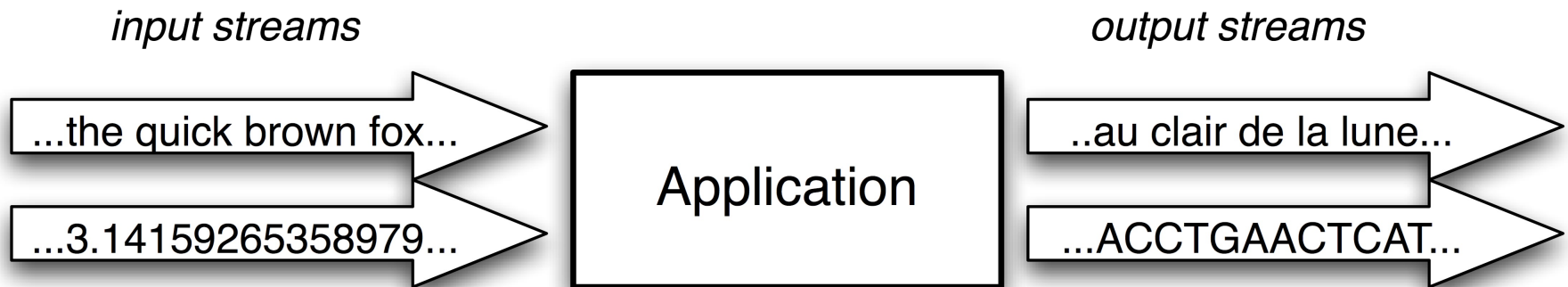


**java.io**

Viewing sequential data as a stream

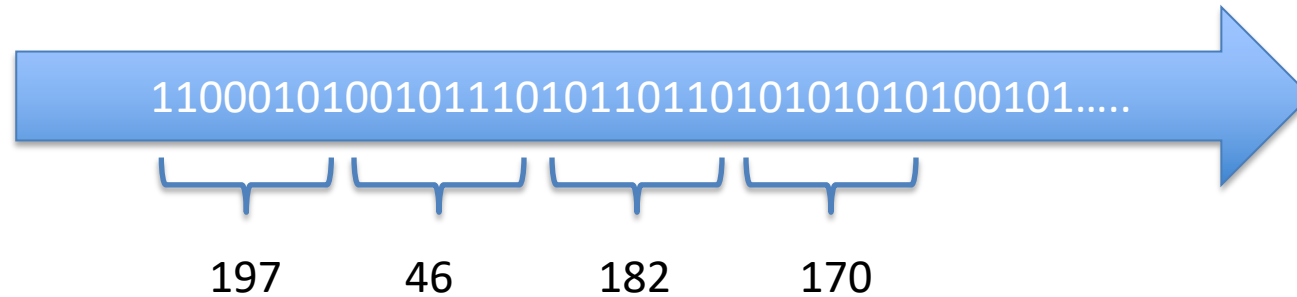
# I/O Streams

- The *stream* abstraction represents a communication channel with the outside world.
  - can be used to read or write a potentially unbounded number of data items (unlike a list)
  - data items are read from or written to a stream one at a time
- The Java I/O library uses subtyping to provide a unified view of disparate data sources and sinks.



# Low-level Streams

- At the lowest level, a stream is a sequence of binary numbers



- The simplest IO classes break up the sequence into 8-bit chunks, called *bytes*. Each byte corresponds to an integer in the range 0 – 255.

# InputStream and OutputStream

- Abstract classes that provide basic operations for the Stream class hierarchy:

```
int read ();           // Reads the next byte of data
void write (int b);  // Writes the byte b to the output
```

- These operations read and write `int` values that represent *bytes*
  - range `0-255` represents a byte value
  - `-1` represents “no more data” (when returned from read)
- `java.io` provides many subclasses for various sources/sinks of data:
  - files, audio devices, strings, byte arrays, serialized objects
- Subclasses also provides rich functionality:
  - encoding, buffering, formatting, filtering

# Binary IO example

```
InputStream fin = new FileInputStream(filename);

int[][] data = new int[width][height];
for (int i=0; i < data.length; i++) {
    for (int j=0; j < data[0].length; j++) {
        int ch = fin.read();
        if (ch == -1) {
            fin.close();
            throw new IOException("File ended early");
        }
        data[j][i] = ch;
    }
}
fin.close();
```

# BufferedInputStream

- Reading one byte at a time can be *slow*!
- Each time a stream is read there is a fixed overhead, plus time proportional to the number of bytes read.\*
  - disk -> operating system -> JVM -> program
  - disk -> operating system -> JVM -> program
  - disk -> operating system -> JVM -> program
- A `BufferedInputStream` presents the same interface to clients, but internally reads many bytes at once into a *buffer* (incurring the fixed overhead only once)

disk -> operating system  
->>>> JVM -> program  
JVM -> program  
JVM -> program  
JVM -> program

## Rule of thumb times to access data:

	actual	for intuition
CPU:	0.5ns	(~ 1 sec)
RAM:	100 ns	(~ 1.6 minutes)
SSD:	150,000 ns	(~ 2.75 days)

\*simplified explanation – the OS, disk, etc., might use caching to speed things up

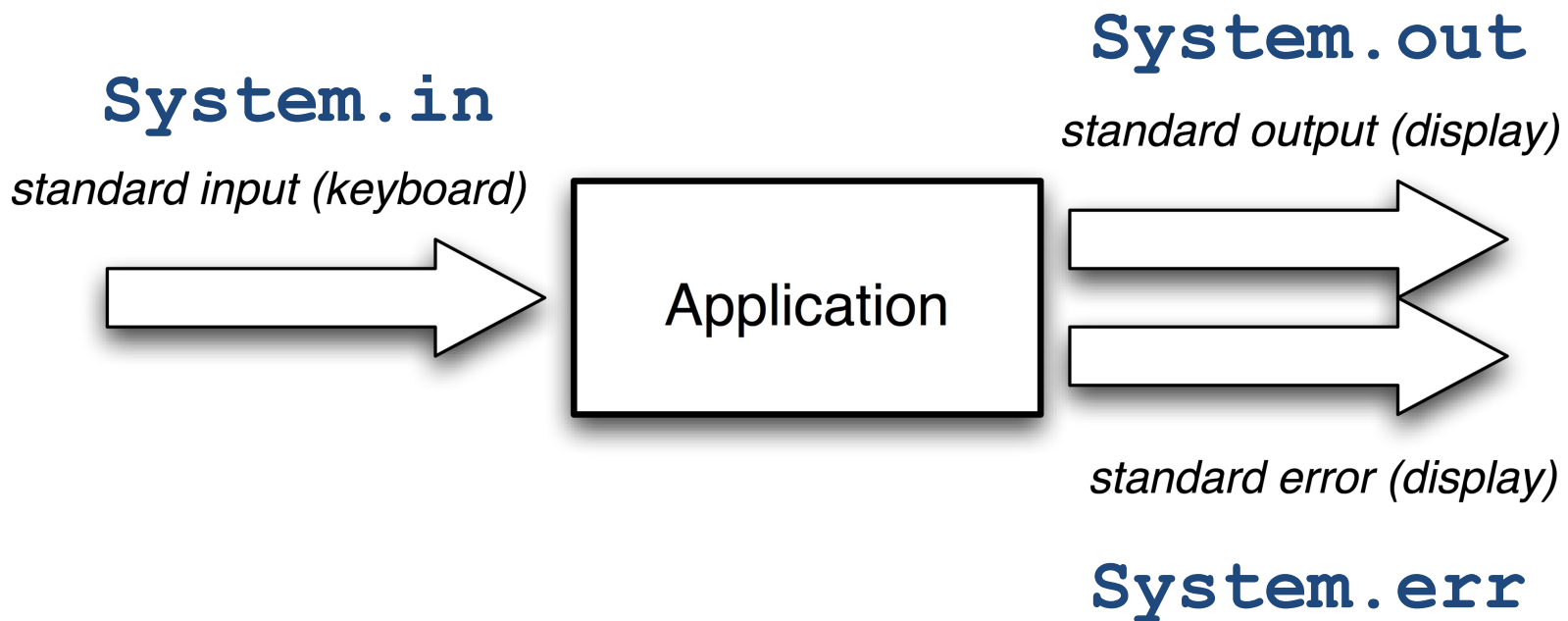
# Buffering Example

```
FileInputStream fin1 = new FileInputStream(filename);
InputStream fin = new BufferedInputStream(fin1);

int[][] data = new int[width][height];
for (int i=0; i < data.length; i++) {
    for (int j=0; j < data[0].length; j++) {
        int ch = fin.read();
        if (ch == -1) {
            fin.close();
            throw new IOException("File ended early");
        }
        data[j][i] = ch;
    }
}
fin.close();
```

# The Standard Java Streams

`java.lang.System` provides an `InputStream` and two standard `PrintStream` objects for doing console I/O.



Note that `System.in`, for example, is a *static member* of the class `System` – this means that the field “`in`” is associated with the *class*, not an *instance* of the class. Recall that static members in Java act like global variables.



# PrintStream Methods

PrintStream adds buffering and binary-conversion methods to OutputStream

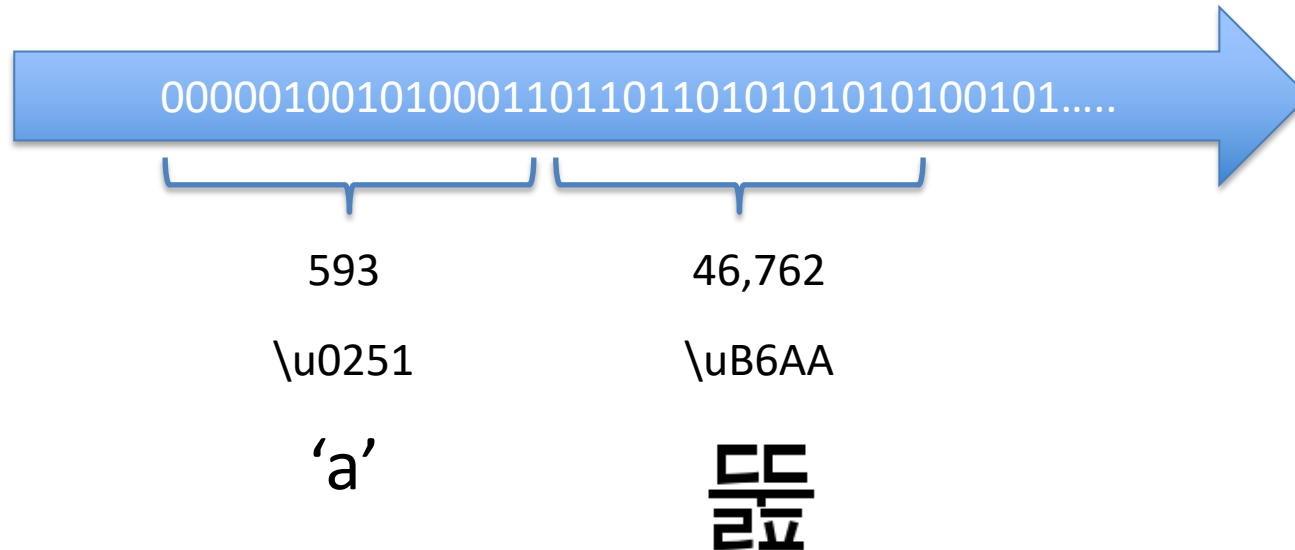
```
void println(boolean b); // write b followed by a new line
void println(String s); // write s followed by a newline
void println();          // write a newline to the stream

void print(String s);    // write s without terminating the line
                        // (output may not appear until the stream is flushed)
void flush();           // actually output characters waiting to be sent
```

- Note the use of *overloading*: there are *multiple* methods called `println`
  - The compiler figures out which one you mean based on the number of arguments, and/or the *static* type of the argument you pass in at the method's call site.
  - The java I/O library uses overloading of constructors pervasively to make it easy to “glue together” the right stream processing routines

# Character based IO

A character stream is a sequence of 16-bit binary numbers



The character-based IO classes break up the sequence into 16-bit chunks, of type `char`. Each character corresponds to a letter (specified by a *character encoding*).

# Reader and Writer

- Similar to the `InputStream` and `OutputStream` classes, including:

```
int read ();           // Reads the next character
void write (int b);   // Writes the char to the output
```

- These operations read and write `int` values that represent *unicode characters*
  - `read` returns an integer in the range 0 to 65535 (*i.e.*, 16 bits)
  - value `-1` represents “no more data” (when returned from `read`)
  - requires an “encoding” (*e.g.*, UTF-8 or UTF-16, set by a `Locale`)
- Like byte streams, the library provides many subclasses of `Reader` and `Writer`. Subclasses also provides rich functionality.
  - use these for portable text I/O
- Gotcha: `System.in`, `System.out`, `System.err` are *byte* streams
  - So, wrap in an `InputStreamReader` / `PrintWriter` if you need unicode console I/O

# Design Example: Histogram.java

A design exercise using java.io and the  
generic collection libraries

(SEE COURSE NOTES FOR THE FULL STORY)

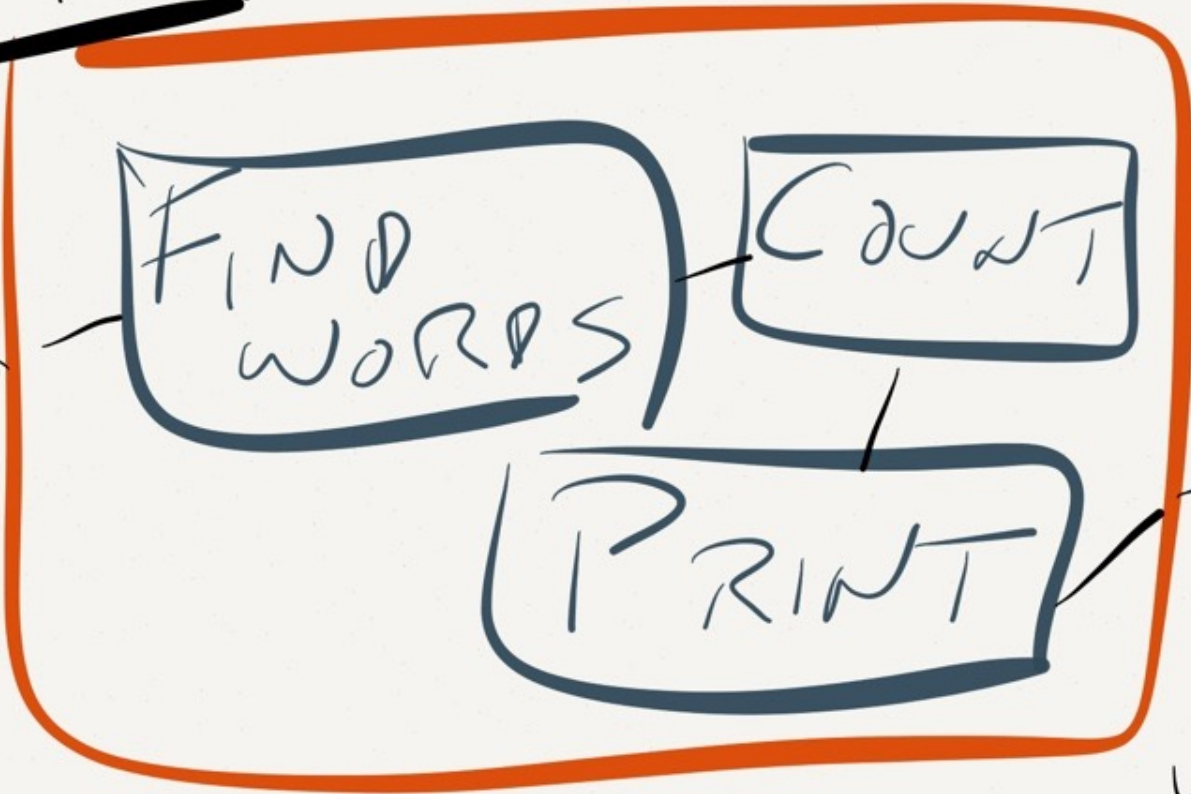
# Problem Statement

Write a program that, given a filename for a text file as input, calculates the frequencies (*i.e.*, number of occurrences) of each distinct word of the file. The program should then print the frequency distribution to the console as a sequence of “word: freq” pairs (one per line).

Histogram result:

The : 1	each : 1	line : 2	should : 1
Write : 1	file : 2	number : 1	text : 1
a : 4	filename : 1	occurrences : 1	that : 1
as : 2	for : 1	of : 4	the : 4
calculates : 1	freq : 1	one : 1	then : 1
command : 1	frequencies : 1	pairs : 1	to : 1
console : 1	frequency : 1	per : 1	word : 2
distinct : 1	given : 1	print : 1	
distribution : 1	i : 1	program : 2	
e : 1	input : 1	sequence : 1	

TEXT FILE



PRINTED HISTOGRAM

# Decompose the problem

- Sub-problems:
  1. How do we iterate through the text file, identifying all of the words?
  2. Once we can produce a stream of words, how do we calculate their frequency?
  3. Once we have calculated the frequencies, how do we print out the result?
- What is the interface between these components?
- Can we test them individually?

# How to produce a stream of words?

1. How do we iterate through the text file, identifying all of the words?

```
public interface Iterator<T> {  
    // returns true if the iteration has more elements  
    public boolean hasNext();  
    // returns the next element in the iteration  
    public T next();  
    // Optional: removes last element returned  
    public void remove();  
}
```

- **Key idea:** Define a class (WordScanner) that implements this interface by reading words from a text file.



# Coding: Histogram.java

WordScanner.java

Histogram.java

# Iterator – hasNext() – First Attempt?

```
@Override
public boolean hasNext() {
    boolean value = true;
    try {
        int c = r.read();
        if (c == -1) {
            value = false;
        }
    } catch (IOException io) {
        System.out.println("IO Exception happened");
    }
    return value;
}
```

### 32: Which combination of the following properties form a useful invariant for the WordScanner fields?



```
public class WordScanner implements Iterator<String> {  
    private Reader r;  
    private int c = -1;  
    // ...  
}
```

Which combination of the following properties form a useful invariant for the WordScanner fields?

- 1. r is not null
- 2. r is null if and only if there is no next word
  
- A. c is 0 if there is no next word and nonzero otherwise
- B. c is -1 if there is no next word and contains the first character of the next word otherwise

1 & A

0%

1 & B

0%

2 & A

0%

2 & B

0%

```
public class WordScanner implements Iterator<String> {  
    private Reader r;  
    private int c = -1;  
    // ...  
}
```

Which combination of the following properties form a useful invariant for the WordScanner fields?

1. r is not null
  2. r is null if and only if there is no next word
- 
- A. c is 0 if there is no next word and nonzero otherwise
  - B. c is -1 if there is no next word and contains the first character of the next word otherwise

ANSWER: 1 & B