

Objectives

- Exceptions
- Files
- Streams

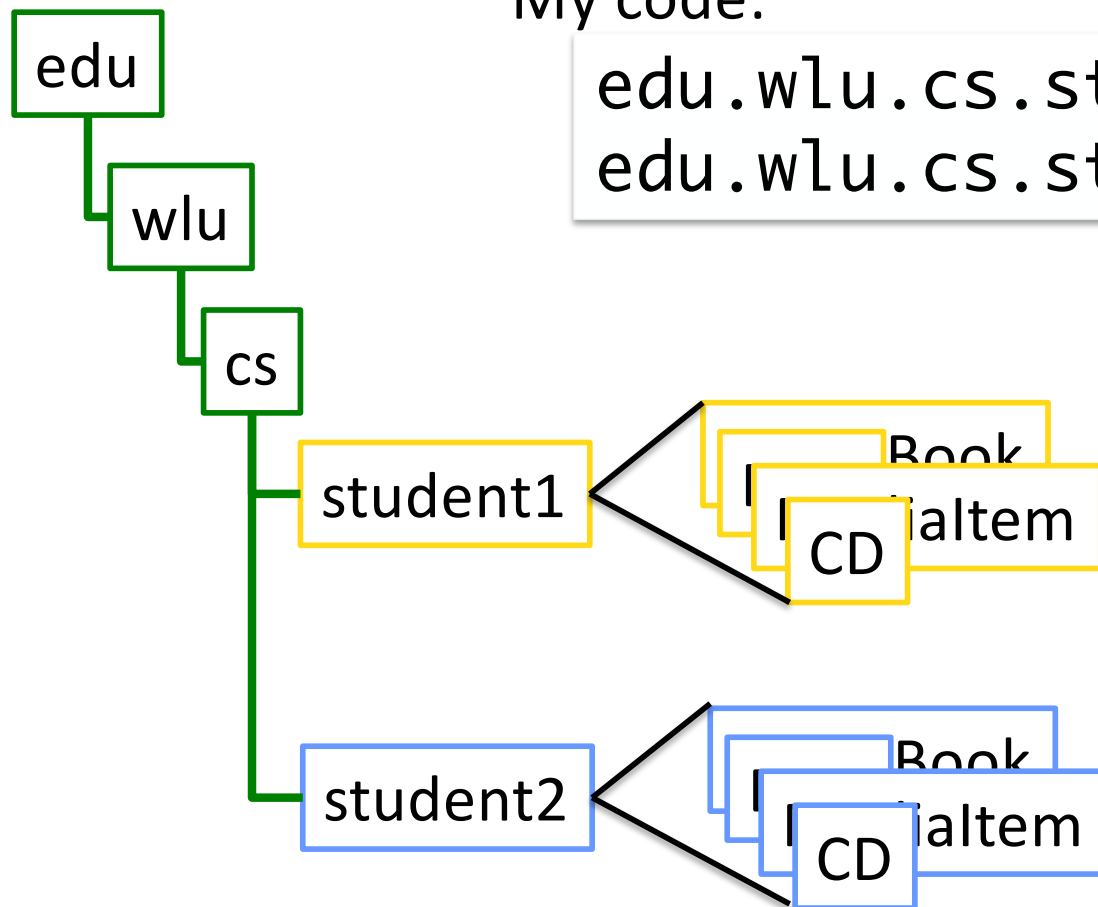
Review

1. What is an Exception?
2. How do we create Exceptions?
3. How do we *advertise* that our method may produce an exception?
4. What are the different categories of exceptions?
 - a) What are examples (i.e., class names) of those categories of exceptions?
5. How do you *handle* an exception? (In Python, this was called “except”)
6. Hypothetical: to grade assignment 7, I am going to write one program that creates objects of each of your [all the students’] classes and call methods on them.
 - But, you all have the same names for your classes!
 - (Because that’s what I told you to do.)
 - How can my code distinguish between the classes?

Review: Packages in Assignment 7

My code:

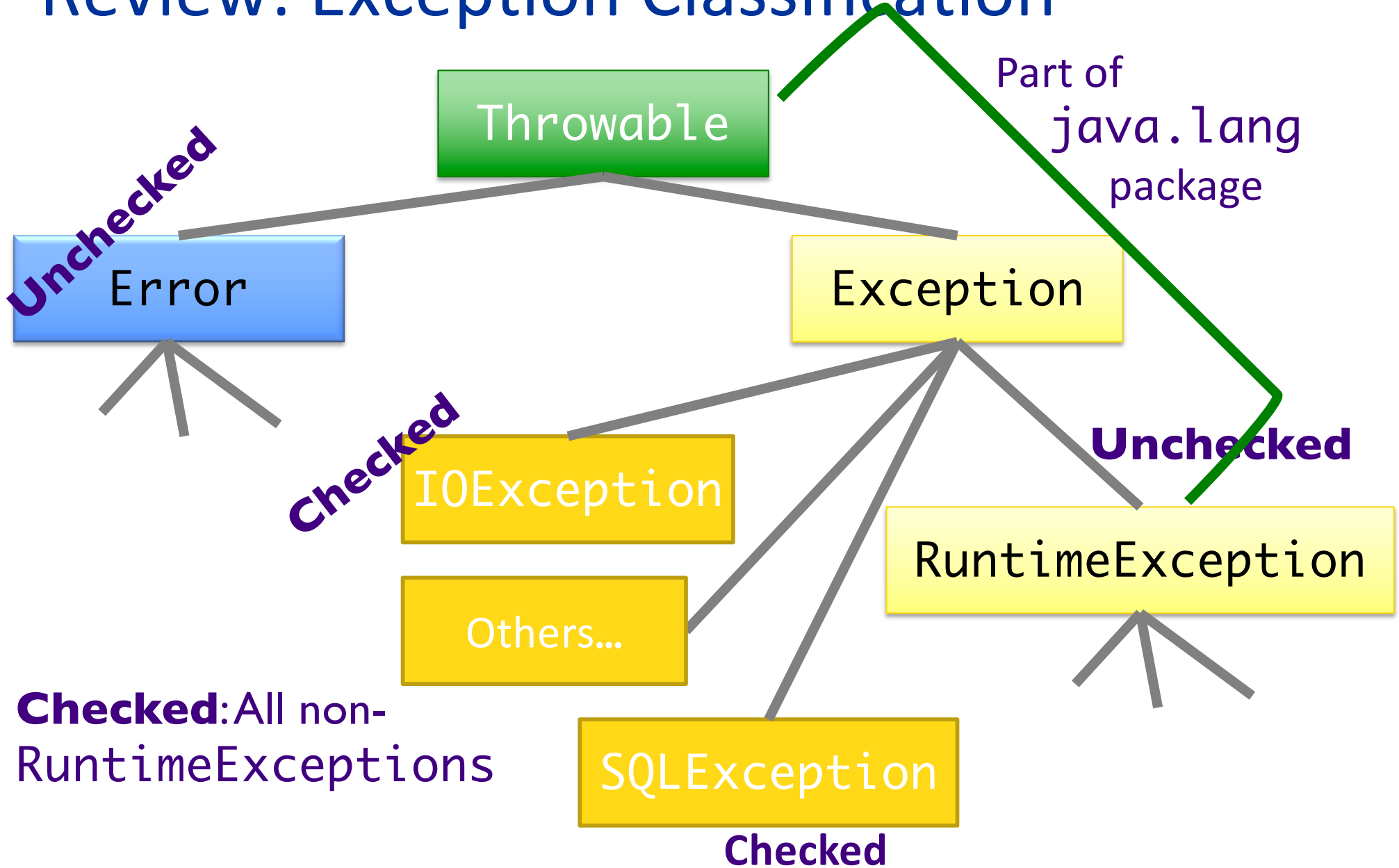
```
edu.wlu.cs.student1.CD cd1 = ...  
edu.wlu.cs.student2.CD cd2 = ...
```



Assignment 7 Review

- Eclipse practice
- Javadocs
 - See what the web pages look like from your comments!

Review: Exception Classification



Review: Categories of Exceptions

Unchecked

- Any exception that derives from `Error` or `RuntimeException`
- Programmer does not necessarily create/handle
- Try to make sure that they don't occur
- Often indicates programmer error
 - E.g., precondition violations, not using API correctly

Checked

- Any other exception
- Programmer creates and handles checked exceptions
- Compiler-enforced checking
 - Improves *reliability**
- For conditions from which caller can reasonably be expected to recover

Review: Types of Unchecked Exceptions

1. Derived from the class `Error`

- Any line of code can generate because it is an internal JVM error
- Don't worry about what to do if this happens

2. Derived from the class `RuntimeException`

- Indicates a bug in the program
- Fix the bug
- Examples: `ArrayOutOfBoundsException`, `NullPointerException`, `ClassCastException`

Review: Throwing An Exception We Created

```
if (grade < 0 || grade > 100) {  
    throw new IllegalArgumentException(  
        "Grade must be between 0 and 100.");  
}
```

1. Create a new object of class **IllegalArgumentException**
 - Class derived from **RuntimeException**
2. **throw** it
 - Method ends at this point
 - Calling method handles exception

Review: Common Exceptions

Name	Purpose
<code>IllegalArgumentException</code>	When caller passes in inappropriate argument
<code>IllegalStateException</code>	Invocation is illegal because of receiving object's state. (Ex: closing a closed window)

- Both inherit from `RuntimeException`
- May seem like these cover everything but only used for certain kinds of illegal arguments and exceptions
- Not used when
 - A null argument passed in; should be a `NullPointerException`
 - Pass in invalid index for an array; should be an `IndexOutOfBoundsException`

Review: Birthday Error Handling Discussion

- Design decision:
 - Since month and day are not independent, should be set together rather than separately
- Check all the error cases before setting the instance variables
 - Don't want an inconsistent birthday after method called
 - Example of Failure Atomicity
- `IllegalArgumentException` is appropriate
 - Programming error
 - Should catch those errors before executing program

CATCHING EXCEPTIONS

Catching Exceptions

- After we throw an exception, some part of program needs to *catch* it
- What does it mean to catch an exception?
 - Program knows how to deal with the situation that caused the exception
 - Handles the problem—hopefully gracefully, without exiting

Try/Catch Block

- The simplest way to catch an exception
- Syntax:

```
try {  
    code;  
    more code;  
}  
catch (ExceptionType e) {  
    error code for ExceptionType;  
}  
catch (ExceptionType2 e) {  
    error code for ExceptionType2;  
}  
...
```

Python equivalent?

Try/Catch Block

- Code in **try** block runs first
- If **try** block completes without an exception, **catch** block(s) are not executed
- If **try** code generates an exception
 - A **catch** block runs
 - Remaining code in **try** block is not executed
- If an exception of a type other than `ExceptionType` is thrown inside **try** block, method exits immediately*
 - Thrown to caller

```
try {  
    code;  
    more code;  
}  
catch (ExceptionType e) {  
    error code for  
    ExceptionType  
}
```

Try/Catch Block

```
try {  
    code;  
    more code;  
}  
catch (ExceptionType1 e) {  
    error code for  
    ExceptionType  
}  
catch (ExceptionType2 e) {  
    error code  
    for ExceptionType2  
}
```

- You can have more than one **catch** block
 - To handle > 1 type of exception
- If exception is not of type **ExceptionType1**, falls to **ExceptionType2**, and so forth
 - Run the first matching **catch** block

Can catch any exception with **Exception e** but won't have customized messages

Try/Catch Example

```
public void read(BufferedReader in) {  
    try {  
        boolean done = false;  
        while (!done) {  
            String line=in.readLine();  
            // above could throw IOException  
            if (line == null)  
                done = true;  
        }  
    }  
    catch (IOException ex) {  
        ex.printStackTrace();  
    }  
}
```

Prints out stack trace to method call
that caused the error.

(Good start during development;
probably should do more.)

Try/Catch Example


```
public void read(BufferedReader in) {
    try {
        boolean done = false;
        while (!done) {
            String line=in.readLine();
            // above could throw IOException
            if (line == null)
                done = true;
        }
    }
    catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

More precise **catch** may help pinpoint error
But could result in messier code

The `finally` Block

- Optional: add a `finally` block after all `catch` blocks
 - Code in `finally` block **always** runs after code in `try` and/or `catch` blocks
 - After `try` block finishes or, if an exception occurs, after the `catch` block finishes
- Allows you to clean up or do maintenance before method ends (one way or the other)
 - E.g., closing files or database connections

```
try {  
    ...  
}  
catch (Exception e) {  
    ...  
}  
finally {  
    ...  
}
```



`FinallyTest.java`

Practice: try/catch/finally Blocks

```
try {  
    statement1;  
    statement2;  
}  
catch (EOFException e) {  
    statement3;  
    statement4;  
}  
finally {  
    statement5;  
}
```

- Which statements run if:
 - Neither *statement1* nor *statement2* throws an exception
 - *statement1* throws an EOFException
 - *statement2* throws an EOFException
 - *statement1* throws an IOException

Practice: try/catch/finally Blocks

```
try {  
    statement1;  
    statement2;  
}  
catch (EOFException e) {  
    statement3;  
    statement4;  
}  
finally {  
    statement5;  
}
```

- Which statements run if:
 - Neither *statement1* nor *statement2* throws an exception
 - 1, 2, 5
 - *statement1* throws an EOFException
 - 1, 3, 4, 5
 - *statement2* throws an EOFException
 - 1, 2, 3, 4, 5
 - *statement1* throws an IOException
 - 1, 5

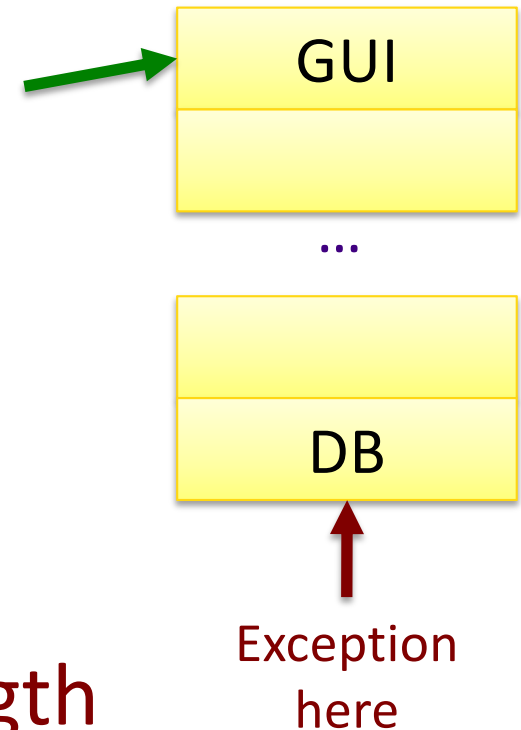
What to do with a Caught Exception?

- Dump the stack after the exception occurs
 - What else can we do?
- Generally, two options:
 1. Catch the exception and recover from it
 2. Pass exception up to whoever called it

To Throw or Catch?

- Problem: lower-level exception propagated up to higher-level code
- Example: user enters account information and gets exception message “field exceeds allowed length in database”
 - Lost context
 - Lower-level detail polluting higher-level API

Handled
here



Solution: higher-levels should catch lower-level exceptions and throw them in terms of higher-level abstraction

Exception Translation

```
try {  
    // Call lower-level abstraction  
}  
catch (LowerLevelException ex) {  
    // TODO: log exception ...  
    throw new HigherLevelException(...);  
}
```

- Special case: Exception Chaining

- When higher-level exception needs info from lower-level exception

```
try {  
    // Call lower-level abstraction  
}  
catch (LowerLevelException cause) {  
    // TODO: log exception ...  
    throw new HigherLevelException(cause);  
}
```

Most standard
Exceptions have this
constructor



Guidelines for Exception Translation

- Try to ensure that lower-level APIs succeed
 - Ex: verify that your parameters satisfy invariants
- Insulate higher-level from lower-level exceptions
 - Handle in some reasonable way
 - Always log problem so admin can check
- If can't do previous two, then use exception translation

Summary: Methods Throwing Exceptions

- API documentation tells you if a method can throw an exception
 - If so, you **must** handle it
- If your method could possibly throw an exception (by generating it or by calling another method that could), advertise it!
 - If you can't handle every error, that's OK...let whoever is calling you worry about it
 - However, they can only handle the error if you advertise the exceptions you can't deal with

Programming with Exceptions

- Exception handling is slow
- Group relevant code together
 - Scope of try/catch block should be small
- Use one big **try** block instead of nesting **try-catch** blocks
 - Speeds up Exception Handling
 - Otherwise, code gets too messy
- Don't ignore exceptions (e.g., **catch** block does nothing)
 - Better to pass them along to higher calls

```
try {  
}  
catch () {  
}  
try {  
}  
catch () {  
}
```

```
try {  
  try {  
  }  
  catch () {  
  }  
}  
catch () {  
}
```

```
try {  
  ...  
  ...  
}  
catch () {  
}
```

Try Block Scope Example

Only this line can throw the exception.

```
public void read(BufferedReader in) {  
    try {  
        boolean done = false;  
        while (!done) {  
            String line=in.readLine();  
            // above could throw IOException  
            if (line == null)  
                done = true;  
        }  
    }  
    catch (IOException ex) {  
    }  
}
```

But all of this code was in **try** block

Why? My considerations:

- In while loop
- Scope of variables
- Readability of code

Try/Catch Block (Lesser) Alternatives

```
public void read(BufferedReader in) {
    boolean done = false;
    try {
        while (!done) {
            String line=in.readLine();
            if (line == null)
                done = true;
        }
    }
    catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

```
public void read(BufferedReader in) {
    boolean done = false;
    while (!done) {
        try {
            String line=in.readLine();
            if (line == null)
                done = true;
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```


Creating Custom Exception Class

- Try to reuse an existing exception
 - Match in name as well as semantics
- If you cannot find a predefined Java `Exception` class that describes your condition, implement a new `Exception` class

FYI... Skipping next 4 slides in class

Creating Custom Exception Class

```
public class FileFormatException extends IOException {  
    public FileFormatException() {  
  
    }  
  
    public FileFormatException(String message) {  
        super(message);  
    }  
  
    // other 2 standard constructors...  
}
```

What happens in this constructor implicitly?

Is this a checked or unchecked exception?

- Can now throw exceptions of type **FileFormatException**

Guidelines for Creating Your Own Exception Classes

- Include accessor methods to get more information about the cause of the exception
 - “failure-capture information”
- Checked or unchecked exception?
 - Checked: *forces* API user to handle BUT more difficult to use API
 - Has to handle all checked exceptions
 - Use checked exception if exceptional condition cannot be prevented by proper use of API *and* API user can take a useful action afterward

Practice: Designing a New Exception Class

- Scenario: When an attempt to make a purchase with a gift card fails because card doesn't have enough money, throw a new exception that you created
- Recall that all Exceptions are Throwable, so they have the methods: `getMessage()`, `printStackTrace()`, `getStackTrace()`

- How would someone else use your class?
- What constructors, additional method(s) may you want to add for your exception class?

Discussion: Benefits of Exceptions

- Been talking about details...
- Why does Java have exceptions as part of the language?
- Why does Java add some features that Python doesn't have?

Benefits of Exceptions

Does NOT mean that error is prevented at compile time—just that we can improve robustness

- Force error checking/handling
 - Otherwise, won't compile
 - Does not guarantee “good” exception handling
- Ease debugging
 - Stack trace
- Separates error-handling code from “regular” code
 - Error code is in catch blocks at end
 - Descriptive messages with exceptions
- Propagate methods up call stack
 - Let whoever “cares” about error handle it
- Group and differentiate error types

FILES

java.io.File Class

- Represents a file or directory
- Provides functionality such as
 - Storage of the file on the disk
 - Determine if a particular file exists
 - When file was last modified
 - Rename file
 - Remove/delete file
 - ...

Making a File Object

- Simplest constructor takes full file name (including path)
 - If don't supply path, Java assumes current directory (.)

```
File myFile = new File("chicken.data");
```

- Creates a `File` *object* representing a file named “chicken.data” in the current directory
- Does ***not*** create a file with this name on disk

Making a File Object

- Simplest constructor takes full file name (including path)
 - If don't supply path, Java assumes current directory (.)

```
File myFile = new File("chicken.data");
```

- Creates a `File` *object* representing a file named “chicken.data” in the current directory
 - Does ***not*** create a file with this name on disk
- Similar to Python:

```
myFile = open("chicken.data")
```

Files, Directories, and Useful Methods

- A `File` object can represent a file **or** a directory
 - Directories are special files in most modern operating systems
- Use `isDirectory()` and/or `isFile()` for type of file `File` object represents
- Use `exists()` method
 - Determines if a file exists on the disk

In Python, these are in the `os.path` module

More File Constructors

- String for the path, String for filename

```
File myFile = new File(  
    "/csdept/courses/cs209/handouts",  
    "chicken.data");
```

- File for directory, String for filename

```
File myDir = new File(  
    "/csdept/courses/cs209/handouts");  
File myFile = new File(myDir, "chicken.data");
```

“Break” any of Java’s Principles?

“Break” any of Java’s Principles?

- Principle of Portability
 - Write and Compile Once, Run Anywhere
- Problem: file paths are OS-specific
- `java.io.File.separator`
 - OSX/Linux: `/`
 - Windows: `\`
- Takeaways:
 - Use relative paths
 - Use configuration files to set paths

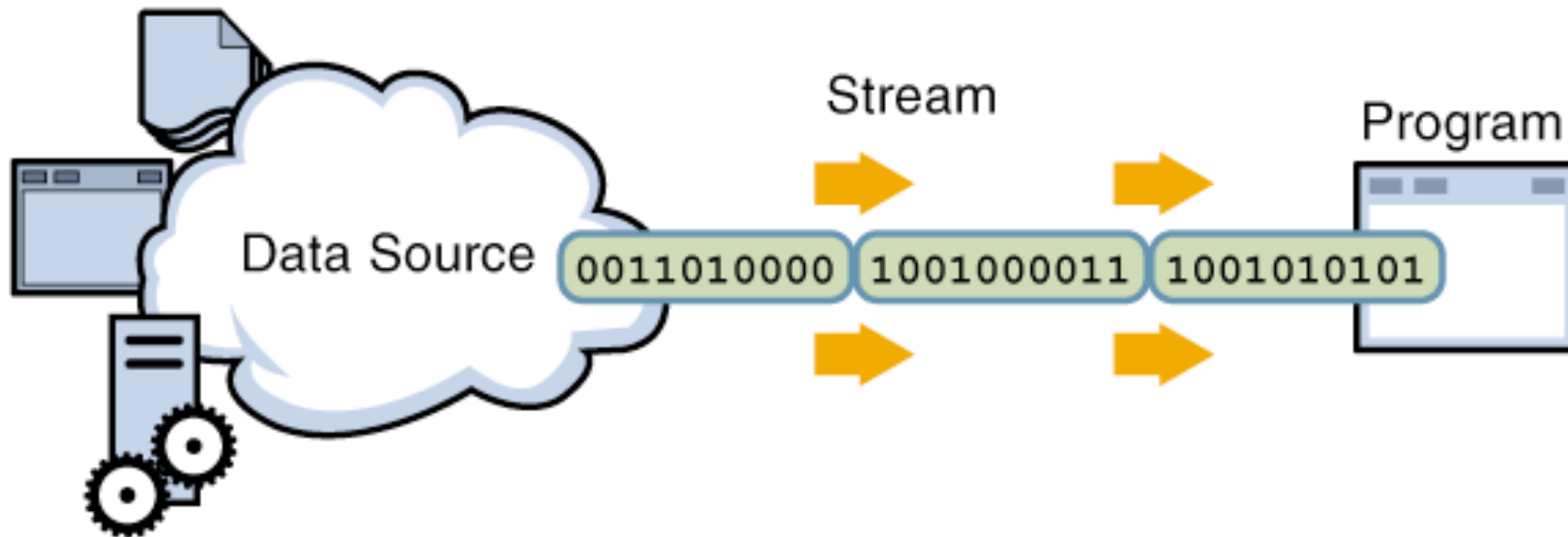
java.io.File Class

- 25+ methods
 - Manipulate files and directories
 - Creating and removing directories
 - Making, renaming, and deleting files
 - Information about file (size, last modified)
 - Creating temporary files
 - ...
- See online API documentation

STREAMS

Streams

Java handles input/output using *streams*, which are sequences of bytes

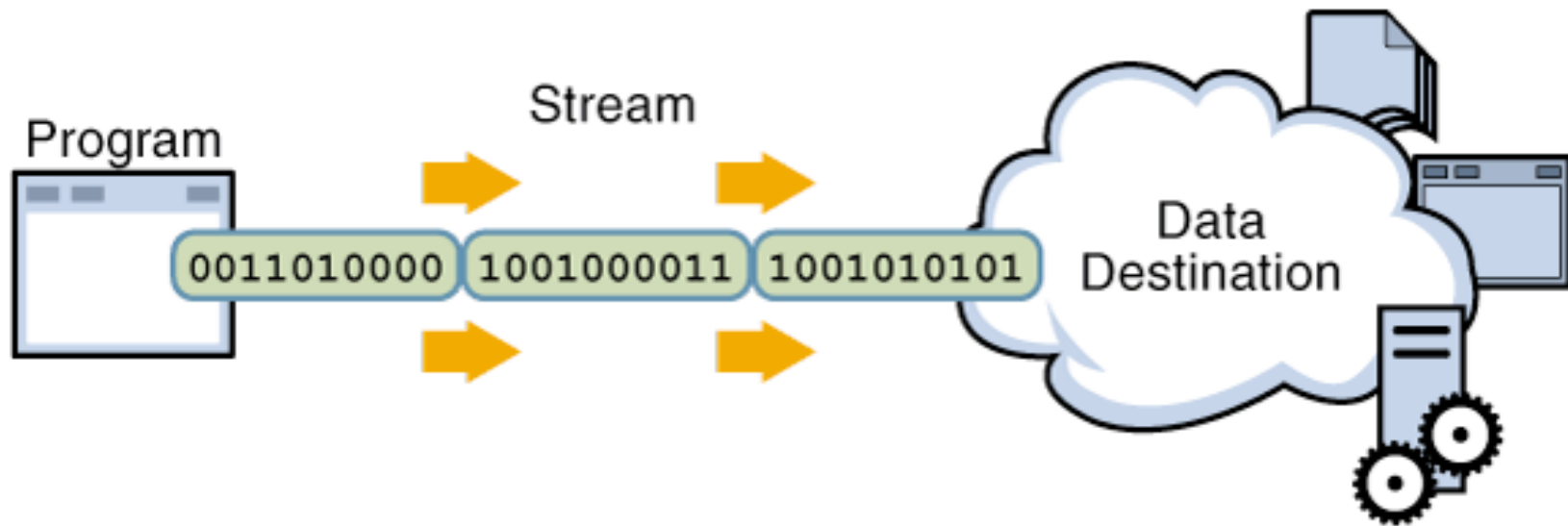


input stream: an object from which we can **read** a **sequence** of bytes

abstract class: `java.io.InputStream`

Streams

Java handles input/output using *streams*, which are sequences of bytes



output stream: an object to which we can **write** a **sequence** of bytes

abstract class: `java.io.OutputStream`

Java Streams

- MANY (80+) types of Java streams
- In `java.io` package
- Why **stream** abstraction?
 - Information stored in different sources is accessed in essentially the same way
 - Example sources: file, on a web server across the network, string
 - Allows same methods to read or write data, regardless of its source
 - Create an `InputStream` or `OutputStream` of the appropriate type

Office hours will end at 12:45 today
Email for other appointments

Exam

- Canvas, timed exam: 70 minutes
 - No class Friday – office hours during that time
 - Open: Friday, 9:30 a.m. – Sunday, 11:59 p.m.
- Open book/notes/slides – but **do not** rely on that
 - NOT open internet
- Prep document online
 - Garbage collection
- 3 sections: Very Short Answer, Short Answer, Coding
- Honor Code
 - No talking about the exam until after September 27