Objectives

- Streams wrap up
- Java Wrap Up
  - Garbage Collection
  - Compiler optimizations
  - Comparing with Python

Review

1. What is a stream?
2. What are 3 different ways to categorize Java stream classes?
3. What design decisions did Java make in creating streams and what are the tradeoffs of those decisions?
    - The design decision could mirror design decisions in other instances/fields/domains. What is an analogy or example of the same design decision?
4. What does the compiler do?
    - How is compiling different from interpreting?

Summary: Streams

- Abstraction: ***streams*** – sequences of data
- Two categories of classes based on type of data they handle
  - Bytes: `InputStream` `OutputStream`
  - Text: `Reader` `Writer`
- Two categories of classes based on their source
  - Data Source (primary source)
  - Filtered (another stream)

Summary: Using Streams

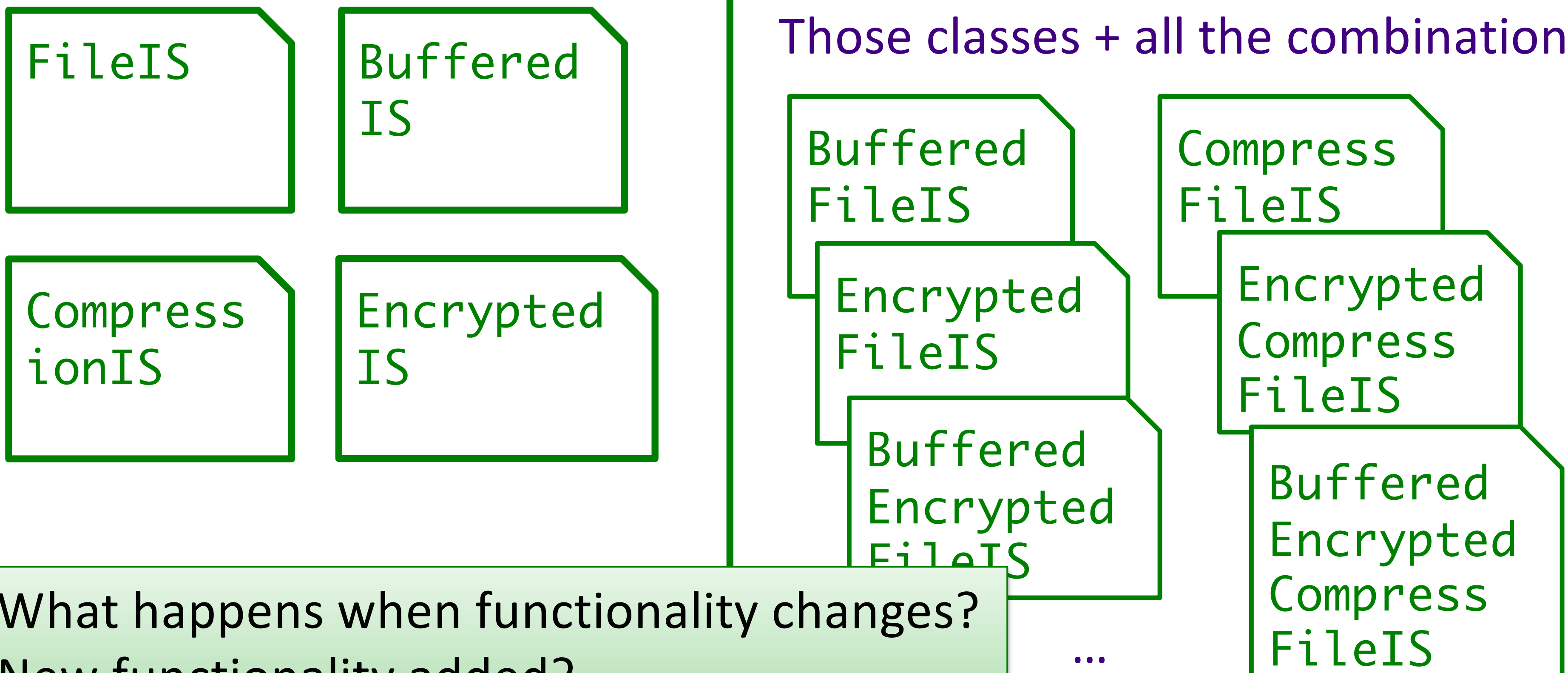
- Can combine streams to get the custom functionality you want
  - Convenience classes for some common combinations
- Development decisions: What do I want this stream to do?
  - What kind of data is it dealing with?
  - What filtering/functionality do I want?
- Select the streams that provide that functionality and connect them (or use convenience class)

Discussion: Stream Design Decisions

Current Design:

Alternative Design:

Those classes + all the combinations



What happens when functionality changes?
New functionality added?

Discussion: Stream Design Decisions

Combine different types of streams
to get functionality you want

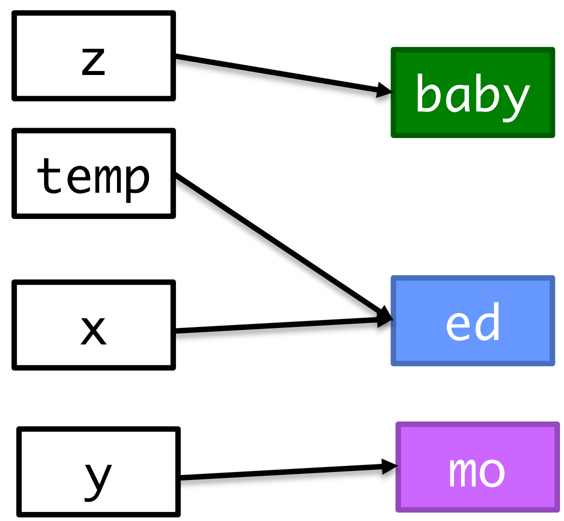
- Alternative: Creating a class for every combination would result in even more classes and a lot of redundant code
  - Consider what is required if some functionality must be updated
  - Tricky for user to pull together various streams BUT also would be hard to find the class you want that has the right combination of functionality

What Happens in This Code?

```
Chicken x, y;
Chicken z = new Chicken("baby", 5, 1.0);
x = new Chicken("ed", 81, 10.3);
y = new Chicken("mo", 63, 6.2);
Chicken temp = x;
x = y;
y = temp;
z = x;
```

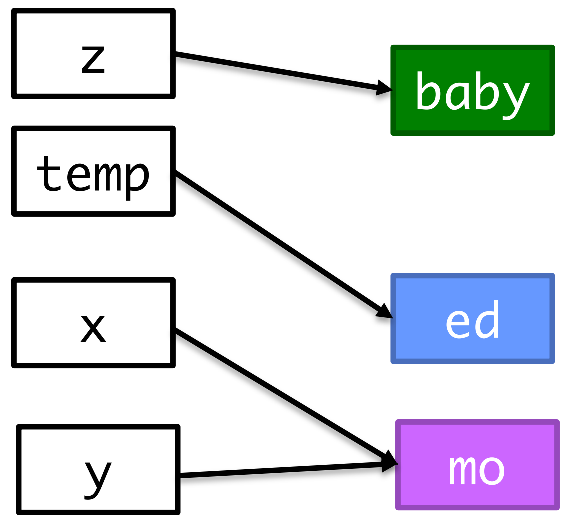
What Happens in This Code?

```
Chicken x, y;
Chicken z = new Chicken("baby", 5, 1.0);
x = new Chicken("ed", 81, 10.3);
y = new Chicken("mo", 63, 6.2);
Chicken temp = x;
x = y;
y = temp;
z = x;
```



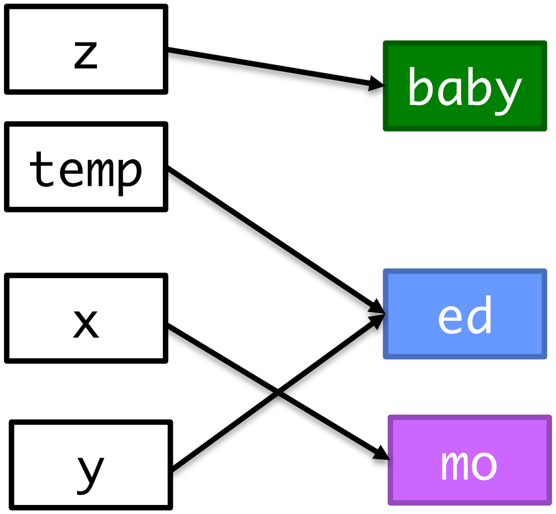
What Happens in This Code?

```
Chicken x, y;
Chicken z = new Chicken("baby", 5, 1.0);
x = new Chicken("ed", 81, 10.3);
y = new Chicken("mo", 63, 6.2);
Chicken temp = x;
x = y;
y = temp;
z = x;
```



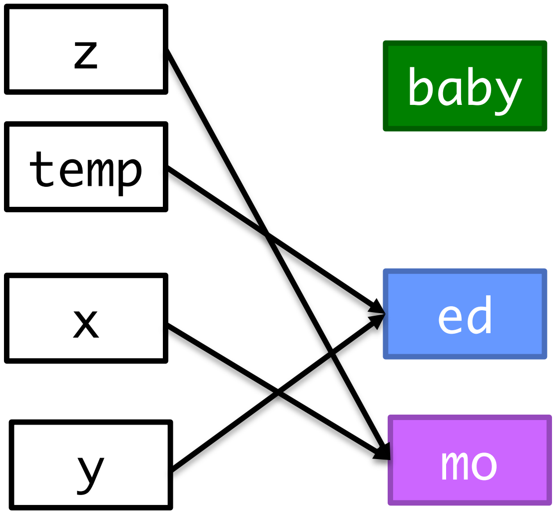
What Happens in This Code?

```
Chicken x, y;
Chicken z = new Chicken("baby", 5, 1.0);
x = new Chicken("ed", 81, 10.3);
y = new Chicken("mo", 63, 6.2);
Chicken temp = x;
x = y;
y = temp;
z = x;
```



What Happens in This Code?

```
Chicken x, y;
Chicken z = new Chicken("baby", 5, 1.0);
x = new Chicken("ed", 81, 10.3);
y = new Chicken("mo", 63, 6.2);
Chicken temp = x;
x = y;
y = temp;
z = x;
```



What Happens in This Code?

```
Chicken x, y;
Chicken z = new Chicken("baby", 5, 1.0);
x = new Chicken("ed", 81, 10.3);
y = new Chicken("mo", 63, 6.2);
Chicken temp = x;
x = y;
y = temp;
z = x;
```

baby

Whoops! Lost "baby" chicken! -- No object variable references it
Memory leak!

Luckily Java has ***garbage collectors*** to clean up the memory leak

GARBAGE COLLECTION

Memory Management

- Early languages (e.g., C): free memory when you're done with it
- In C++ and some other OOP languages, classes have explicit ***destructor*** methods that run when an object is no longer in scope
- Java provides ***automatic garbage collection***
  - Reclaims memory allocated for objects that are no longer referenced

Garbage Collector

- Garbage collector is low-priority thread
  - Or runs when available memory gets tight
  - i.e., it doesn't necessarily immediately free memory
- Before GC can clean up an object, the object may have opened resources
  - Ex: generated temp files or open network connections that should be deleted/closed first
- GC calls object's `finalize`() method
  - Object's chance to clean up resources

`finalize()`

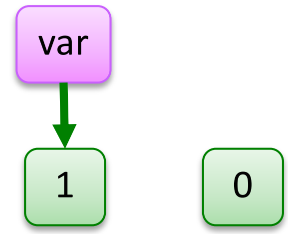
- Inherited from `java.lang.Object`
- Called before garbage collector sweeps away an object and reclaims its memory
- Should not be used for reclaiming resources
  - i.e., *close resources as soon as possible*
  - Why?
    - *When* method is called is not deterministic or consistent
    - Only know it will run sometime before garbage collection
- Clean up anything that cannot be atomically cleaned up by the garbage collector
  - Close file handles, network connections, database connections, etc.
- Note: no finalizer chaining
  - Must explicitly call parent object's `finalize` method

Alternatives to `finalize`

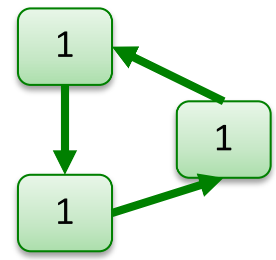
- Recall: unknown when `finalize` will execute—or *if* it will execute
  - Also *heavy performance cost*
- Solution: create your own terminating method
  - User of class terminates when done using object
- Examples: `File`'s or `Scanner`'s `close` method
- May still want `finalize()` as a safety net if user didn't call the terminate method
  - Log a warning message so user knows error in code

Do you know what Python does?

Python Garbage Collection



- Python also does garbage collection
- Python does ***reference counting***
  - On each reference/dereference, update the number of references to the object
    - Can't handle reference cycles



- Python also does ***generational garbage collection*** to handle reference cycles
- Tradeoffs with Java's Garbage Collection
  - Synchronous (not asynchronous) process (i.e., slows down execution)
  - Cheaper memory costs than Java for keeping track of what can be garbage collected

Discussion: Benefits and limitations of garbage collection?

Garbage Collection

Benefits

- Programmer doesn't need to worry about memory management
- Cleans up unused memory automatically, eventually
- Programmer can never release memory that is then accessed (a.k.a. seg faults)

Drawbacks

- Programmer doesn't worry about memory management
    - May not be as careful to avoid memory leaks
- Memory could be cleaned up sooner
- Requires resources (CPU, memory) to keep track of memory
- Slows program execution

Garbage Collection

Benefits

- Programmer doesn't need to worry about memory management
- Cleans up unused memory automatically, eventually
- Programmer can never release memory that is then accessed

Drawbacks

- Programmer doesn't worry about memory management
  - May not be as careful to avoid memory leaks
- Memory could be cleaned up sooner
- Requires resources (CPU, memory) to keep track of memory
- Slows program execution

- Generally, programmer time is more valuable than computer resources.
- Generally, less buggy code is preferred to more efficient code.

COMPILATION

Review

- What does the compiler do?
- How is compiling different from interpreting?

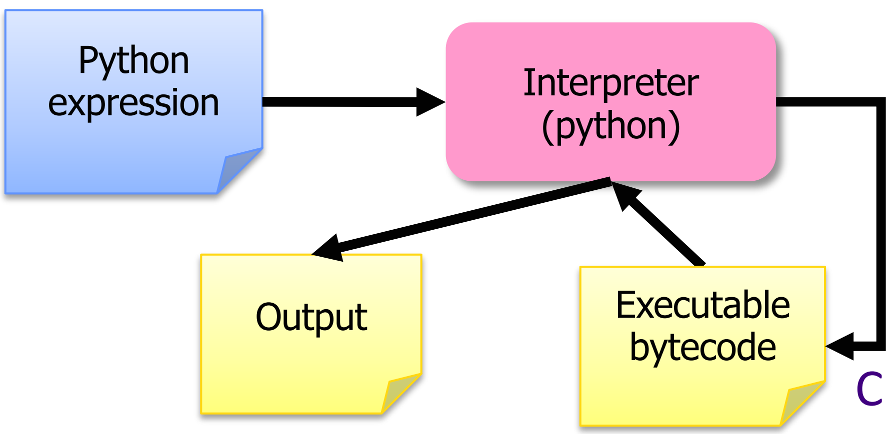
Compiling

- Translates high-level programming language to machine code or byte code
  - Java: .java → .class == bytecode
  - Holistic view of the program
- Compiler optimization techniques
  - Generate *efficient* bytecode/machine code
  - In Java: static typing for additional gains
- Can execute generated code multiple times
  - Performance gain
  - Interpreted → have to re-verify the code each time executed

Python Interpreter

(not pure interpreting)

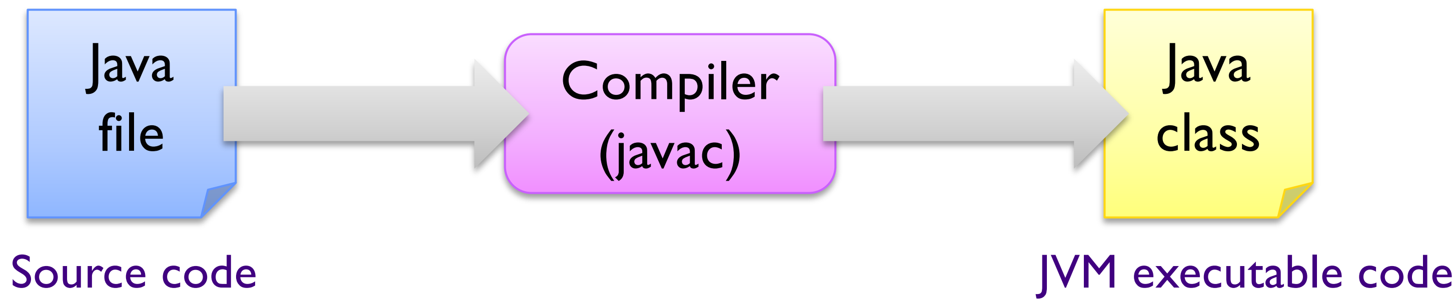
1. Validates Python programming language expression(s)
   - Enforces Python syntax rules
   - Reports syntax errors
2. Executes expression(s)



Only if no syntax errors

Caches compiled byte code in __pycache__

Java Compiler



- Lexical analysis, parsing, semantic analysis, *code generation*, and *code optimization*
- Code optimization: dead code eliminator, inline expansion, constant propagation, …

Compiled vs Interpreted Languages

In pure forms

Compiled

- – Spends a lot of time analyzing and processing the program
- • Resulting executable is some form of machine- specific binary code
- • Computer hardware interprets (executes) resulting code
- ✓ Program execution is fast
  - ➢ Efficient machine/byte code generation
  - ➢ Performance gains

Interpreted

- ✓ Relatively little time spent analyzing and processing the program
- • Resulting code is some sort of intermediate code
- • Another program interprets resulting code
- – Program execution is relatively slow
- ✓ Faster development/prototyping

Compiler Optimization Examples*

- What is the optimization?
  - How is the resulting code more efficient?
- For each optimization approach, generally,
  - should you make these optimizations yourself?
  - Or, is it something that only the compiler should do?
  - Key question: what is likely to change?

*Not literally what the code optimizations look like
- Optimizations are in byte code
- CSCI210 may help illuminate why these decrease runtime

Compiler Optimization: Example 1

Original:

```
for(int i = 0; i < 10; i++ ) {
        int j = 10;
        System.out.println(i + ", " + j);
}
```

Optimization 1

```
int j = 10;
for(int i = 0; i < 10; i++ ) {
        System.out.println(i + ", " + j);
}
```

Optimization 2

```
for(int i = 0; i < 10; i++ ) {
        System.out.println(i + ", " + 10);
}
```

Compiler Optimization: Example 2

Original:

```
for( int i = 0; i < 10; i++ ) {
        if( i == 0 ) {
                System.out.println("Do this");
        }
        else {
                System.out.println("Do that");
        }
}
```

Optimization 1

```
System.out.println("Do this");

for( int i = 1; i < 10; i++ ) {
        System.out.println("Do that");
}
```

Optimization 2

```
System.out.println("Do this");
System.out.println("Do that");
System.out.println("Do that");
System.out.println("Do that");
…
```

Compiler Optimization: Example 3

Original:

```
public void f(int i) {
    a[0] = i + 0;
    a[1] = i * 0;
    a[2] = i - i;
    a[3] = 1 + i + 1;
}
```

Optimization 1

```
public void f(int i) {
  a[0] = i;
  a[1] = 0;
  a[2] = 0;
  a[3] = i + 2;
}
```

Compiler Optimization: Example 4

Original:

```
int add(int x, int y) {
    return x + y;
}

int sub(int x, int y) {
    return add(x, -y);
}
```

add method stays the same

Optimization 1

```
int sub(int x, int y) {
    return x + -y;
}
```

Optimization 2

```
int sub(int x, int y) {
    return x - y;
}
```

Compiler Optimization: Example 5

```java
class Parent {
        void final f() {
            System.out.println("f");
        }
}
```

```java
for( Parent p : parentArray ) {
        p.f(); // check p's actual type at runtime
               // and execute its method f
}
```

Optimization:

```java
for( Parent p : parentArray ) {
        System.out.println("f");
}
```

Compiler Tradeoffs

- Upfront costs
  - Searching for optimizations
  - Make optimizations
    - Typically not Big-O efficiency improvements (unless program is written really inefficiently)
  - Iterative process: make optimizations and then look for more optimizations
- Improved runtime
  - Expect executed many more times than compiled

Looking Ahead

- Monday: Assignment 5
  - People are having trouble with their Eclipse set up, so start soon if you haven’t already!