Objectives

- Streams wrap up
- Compiling
  - VS Interpreting
  - Optimizations
- Language Comparison

Review

1. What is a stream?
   - How does the abstraction of streams help in developing our code?
2. What are 3 different ways to categorize Java stream classes?
3. Scenario: you know you need to use a stream. What questions will you ask to figure out how to use/pick/create the stream?
4. What does the compiler do?
   - How is compiling different from interpreting?

Summary: Streams

- Abstraction: ***streams*** – sequences of data
  - independent of the data source
- 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 is the source/destination of the data?
  - What filtering/functionality do I want?
- Select the streams that provide that functionality and connect them (or use convenience class)

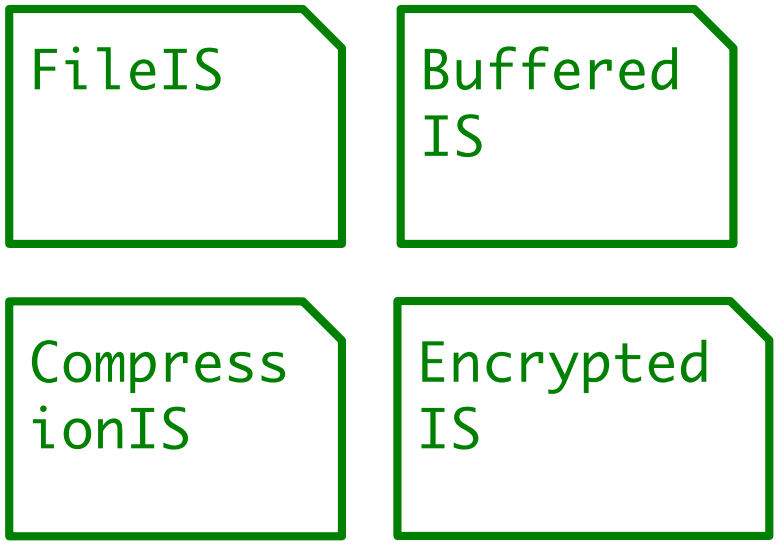
Discussion: Stream Design Decisions

- Java's Streams
  - Combine different types of streams to get functionality you want
  - Provide convenience classes for common functionality

- What are the tradeoffs for this design decision?
  - What would the alternatives be?
  - Consider if you maintained the Java libraries
  - Consider as a user of those Java libraries
- The design decision mirrors design decisions in other instances/fields/domains. What is an analogy or example of the same design decision?

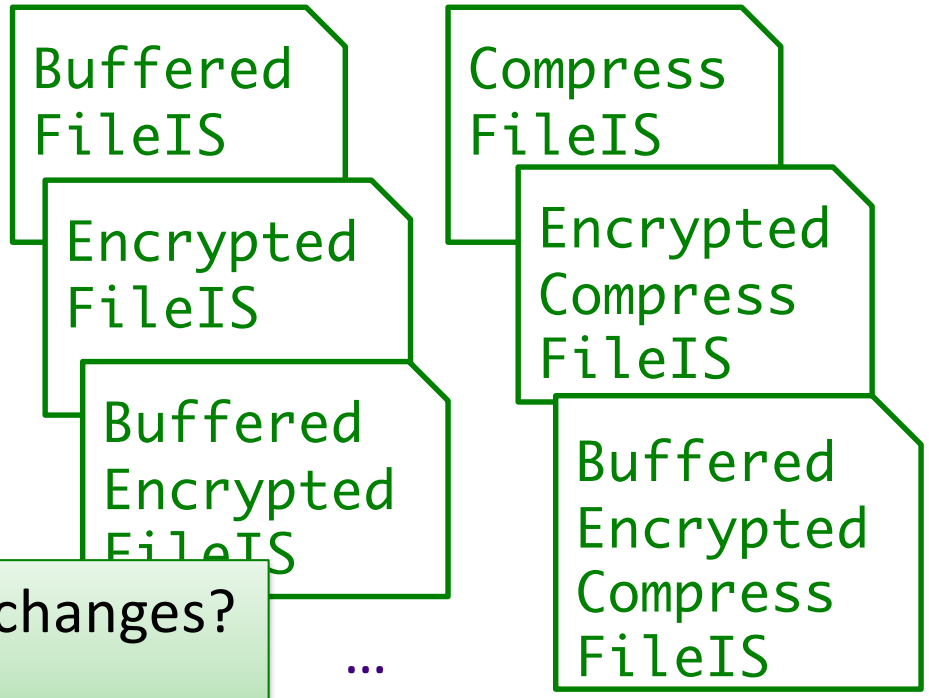
Discussion: Stream Design Decisions

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

COMPILATION

Review

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

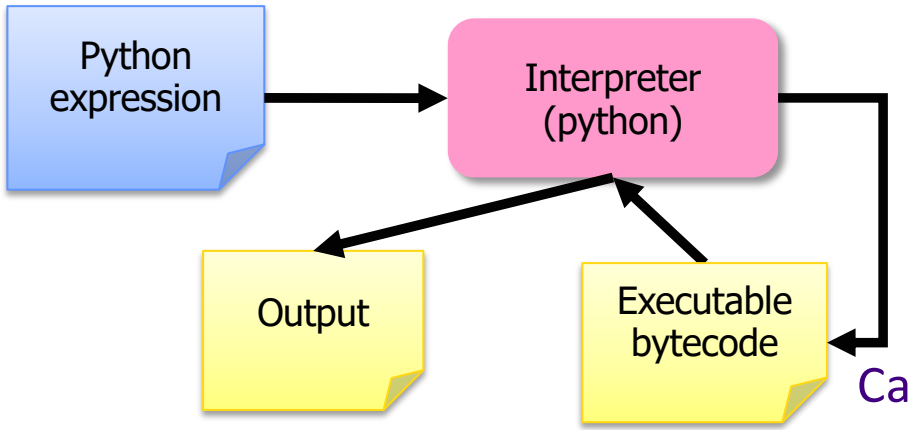
Compiling

- Translates high-level programming language to machine code or byte code
  - C/C++ → machine code
  - 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

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

```
for( Parent p : parentArray ) {
        p.f();
}
```

Optimization:

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

Compiler Optimization: Example 5

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

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

Optimization:

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

Different Perspectives on the Program

To the Compiler

- This is my one shot to validate the program and optimize it!

To You/Developer

- The long view: I am compiling the program now, but I could change the program later.
  - It should be easy to update the program; otherwise, I could introduce bugs.

Compiler Tradeoffs

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