Objectives

- Java Wrap Up
  - Compiler optimizations
  - Comparing with Python

1

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

2

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

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

5

COMPILATION

6

Review

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

7

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
  - Examples: get rid of unused local variables, transform loops, inline method calls
  - 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

8

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

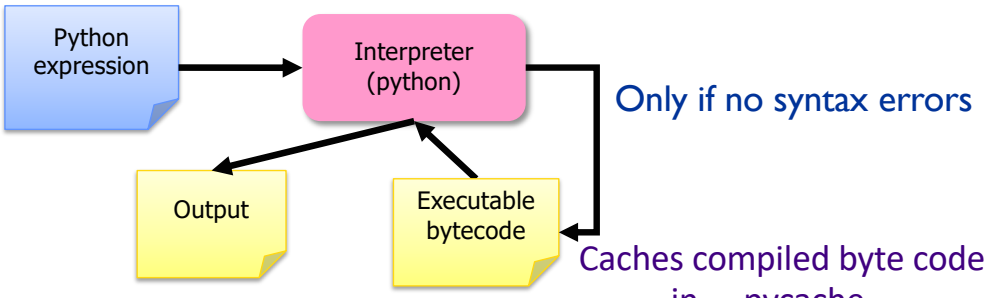
Interpreted

- ✓ 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

Python Interpreter

(not pure interpreting)

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



Java Compiler

Source code

Executable code

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

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(); // check p's actual type at runtime
                 // and execute its method f
}
```

Optimization:

```
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

LANGUAGE COMPARISON

19

Language Comparison

Java **Python**

1) Focus on their characteristics (just the facts, not tradeoffs)
2) Pros and cons, preferences

20

Language Comparison

Java

- Entirely Object-oriented*
  - Procedural
  - Functional - newer
- Statically, strongly typed
- Compiled

Python

- Object-oriented
  - Also procedural and functional programming
- Dynamically, strongly typed
- Interpreted

Pros and cons of using each?

Rest of the Semester

- Shift from learning Java, specifically, to learning how to develop software (abstractly) with Java as our implementation/example
- Why Java?
  - Popular language
  - Many frameworks and tools for Java
  - Java's structure allows for strict adherence to design techniques
- Just a start on Java
  - You'll need to continue learning more Java on your own

Looking Ahead

- Friday: Assignment 5