# Objectives

- Compiling
- Java vs Python
- Software Development

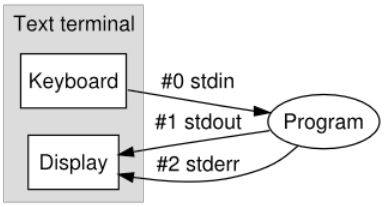




# Review

1. What is a stream?
2. What are 3 different ways to categorize Java stream classes?
3. Java provides a bunch of classes that we can combine to get the functionality we want. What are the tradeoffs of that design decision?
4. What are the 3 standard streams? (not Java-specific)
   - How do we refer to those streams in Java?

# Review: Standard Streams




- Preconnected streams
  - Standard Out: stdout
  - Standard In: stdin
  - *Standard Error: stderr*
    - For error messages and diagnostics
    - In Java: `System.err`
- Helpful to have separate streams for output and error messages
  - Can save outputs in two different files, e.g., error.log vs output.log
  - Eclipse (and other IDEs) differentiates between output (black text) and error (red text)

```
java StandardStreamsExample 1> out 2> err
```





# Output in Python 3

- `print`'s documentation
  - Defaults printing to stdout:

```
print(...)
    print(value, ..., sep=' ', end='\n',
          file=sys.stdout, flush=False)
```

- To print to stdout:

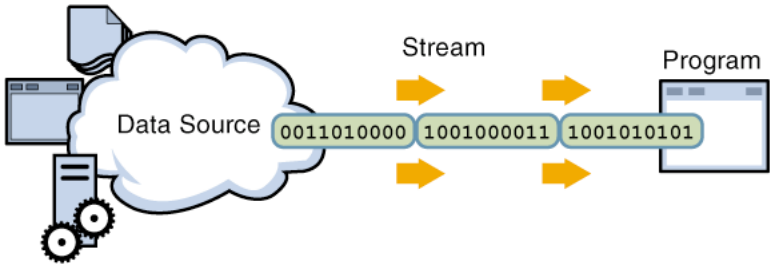
```
print("Goes to standard out by default")
```

- To print to stderr, import sys module

```
print("Goes to standard error", file=sys.stderr)
```

## Review: 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`



5

## Review: Stream Categories

1. Two categories based on flow of stream
   1. Input
   2. Output
2. Two categories of classes based on type of data they handle
   1. Bytes: `InputStream OutputStream`
   2. Text: `Reader Writer`
3. Two categories of classes based on their source
   1. Data Source (primary source)
   2. Filtered (another stream)



6

## Review: Connected Streams

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





## Closing Streams

- Through experimentation: Closing the outermost connected stream will close the inner (wrapped) stream:

```
FileInputStream fis = new FileInputStream("myfile.dat");

BufferedReader br = new BufferedReader(new InputStreamReader(fis));
br.close();

// calling a reading-related method on the fis stream will throw an
// exception because the stream is already closed.
fis.available();
```

- Closing an already closed stream does not throw an error

## Summary: 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

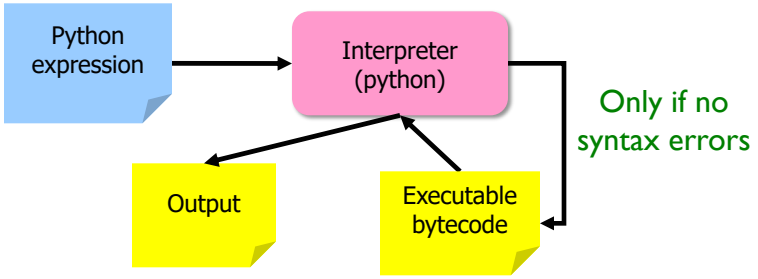
- ✓ Relatively little time spent analyzing and processing the program
- • Resulting code is some sort of intermediate code
- • Another program interprets resulting code
- − Program execution is relatively slow
- ✓ Faster development/prototyping





## Python Interpreter

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



# Review: Compiling

- Translates high-level programming language to machine code or byte code
  - Java: .java → .class == bytecode
- 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





# Compiler Optimization Examples*

- What is the optimization?
  - How does it make the code more efficient?
- For each optimization
  - Should you transform the code yourself to do that optimization?
  - Or, is it something that only the compiler should do?

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

# Compiler Optimization Examples

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 Examples

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

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

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

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

## Compiler Optimization Examples

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

- Why is the code written like this? It seems silly!
- Likely after some previous optimizations
  - Ex: know variable is a constant

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





## Compiler Optimization Examples

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

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

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

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

# Compiler Optimization Examples

```
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-Oh efficiency improvements (unless program is really inefficient)
- Improved runtime
  - Expect executed many more times than compiled

## Should You Apply the Optimization?

- Your priority: keeping code abstract to make it easier to change
- If you can apply the optimization without making the code harder to change, you should do it



19

## LANGUAGE COMPARISON



20

## Language Comparison

**Java**

**Python**



21

## Language Comparison

**Java**

- Entirely Object-oriented*
  - Functional programming mimicked through using just static methods within a class
- Statically, strongly typed
- Compiled

**Python**

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

Pros and cons of using each?



22

## Student's Analogy

I think of Python like writing bullet points whereas Java is like writing full sentences.
Defining an idea with bullet points (aka Python) is doable in fewer words and is more 'high-level' but has to be interpreted as a complete idea.

Full sentences (aka Java) have more overhead (capital letters, periods, etc.) but can be compiled into a complete paragraph and can be read out loud (executed) more efficiently





## 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

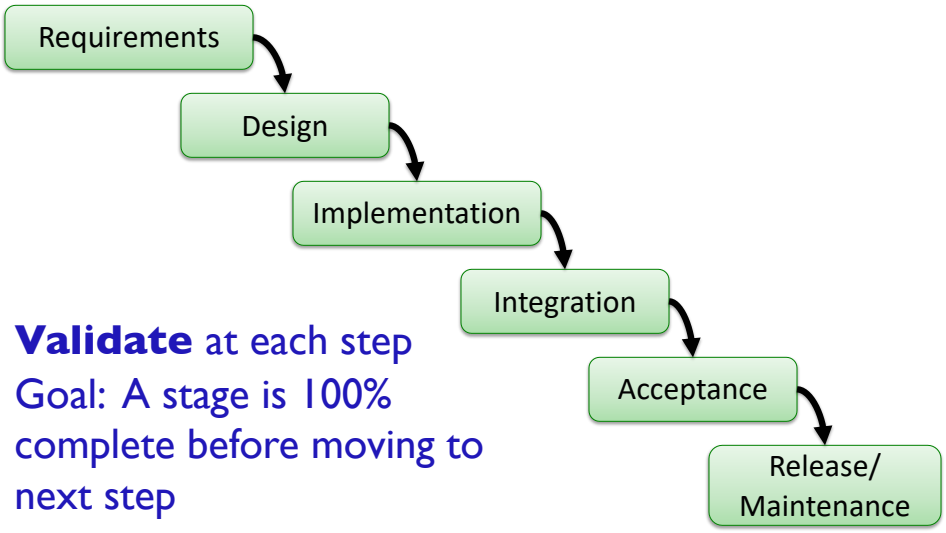
"There is no single development, in either technology or in management technique, that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity." – Fred Brooks

# SOFTWARE DEVELOPMENT



25

## Traditional Software Engineering Process: Waterfall Model

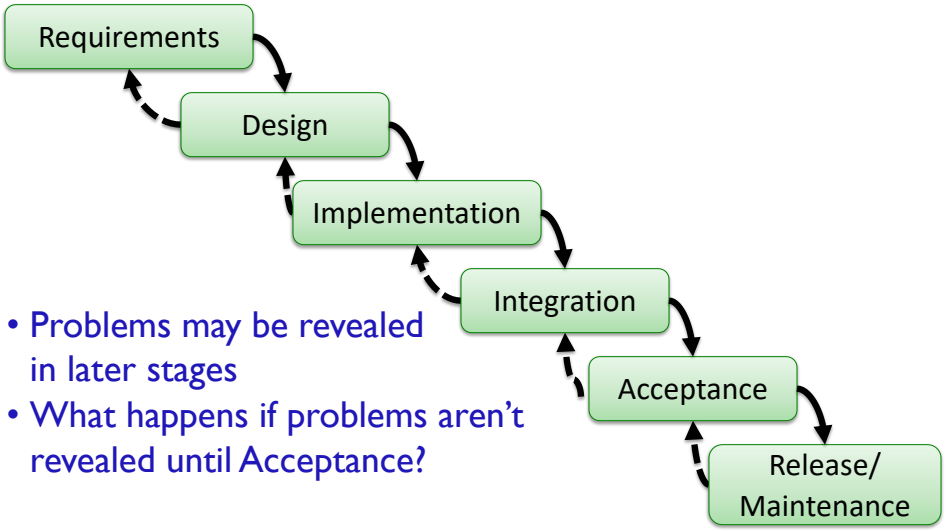



**Validate** at each step
Goal: A stage is 100% complete before moving to next step



28

# Feedback in Waterfall Model

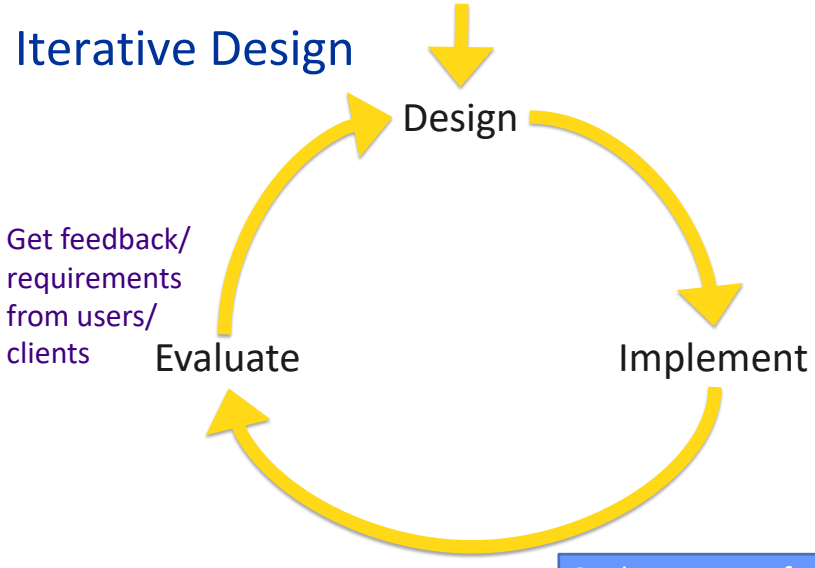



- Problems may be revealed in later stages
- What happens if problems aren't revealed until Acceptance?



29

# Iterative Design




Get feedback/ requirements from users/ clients

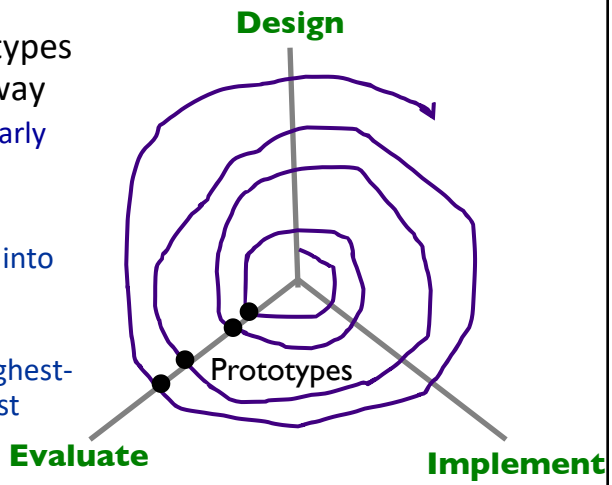
Goals: Frequent feedback
→ Identify problems early
→ Higher quality product



30

# Spiral Model

- Idea: smaller prototypes to test/fix/throw away
  - Finding problems early costs less
- In general…
  - Break functionality into smaller pieces
  - Implement most depended-on or highest-priority features first




[Boehm 86]

Radial dimension: cost





# Looking Ahead

- Review slides about testing, JUnit before class
  - Canvas quiz
- Goal: Hands-on lab in class on Friday