Objectives

- Streams
- Standard Error
- Java wrap up
  - Jar files
  - Classpath
  - Compiling

Review

1. What are the different categories of exceptions?
   a) What are examples (i.e., class names) of those categories of exceptions?
2. If your code calls a method that can throw an exception, how can you handle it?
3. What is a stream?
4. What are 3 different ways to categorize Java stream classes?

Fun Fact: Python also has `finally`

```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
```

https://docs.python.org/3/tutorial/errors.html

Fun Fact: Python also has `finally`

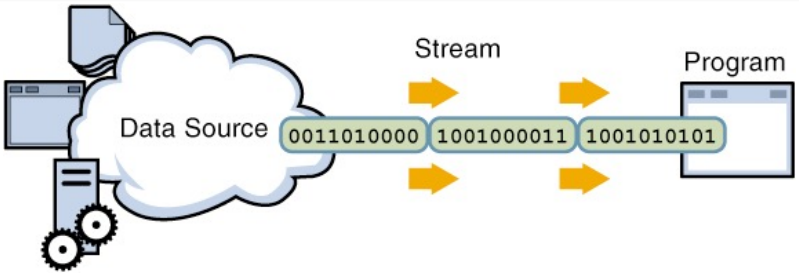
```
def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionErro
        print("division by
    else:
        print("result is",
    finally:
        print("executing fi
```

```
>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 3, in divide
TypeError: unsupported operand
type(s) for /: 'str' and 'str'
```

https://docs.python.org

Review: Streams

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

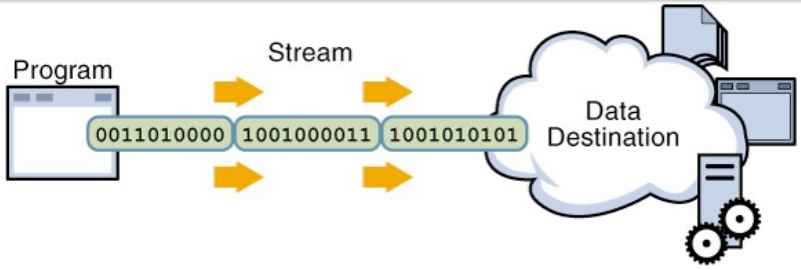


input stream: an object from which we can ***read*** a sequence of bytes

abstract class: `java.io.InputStream`

Review: Streams

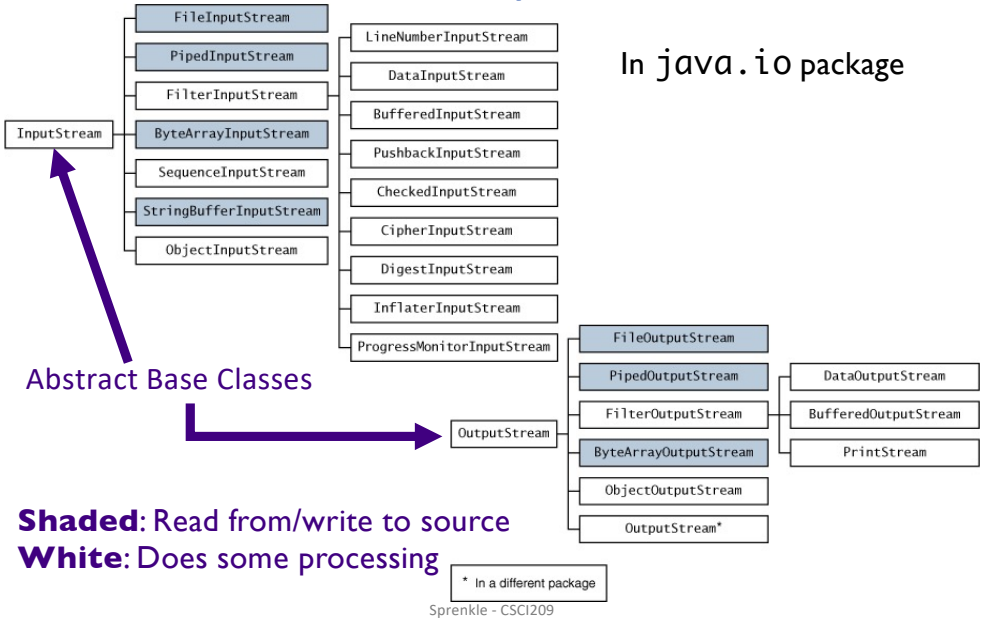
Java handles input/output using ***streams***, which are sequences of bytes



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

abstract class: `java.io.OutputStream`

Byte Streams: For Binary Data

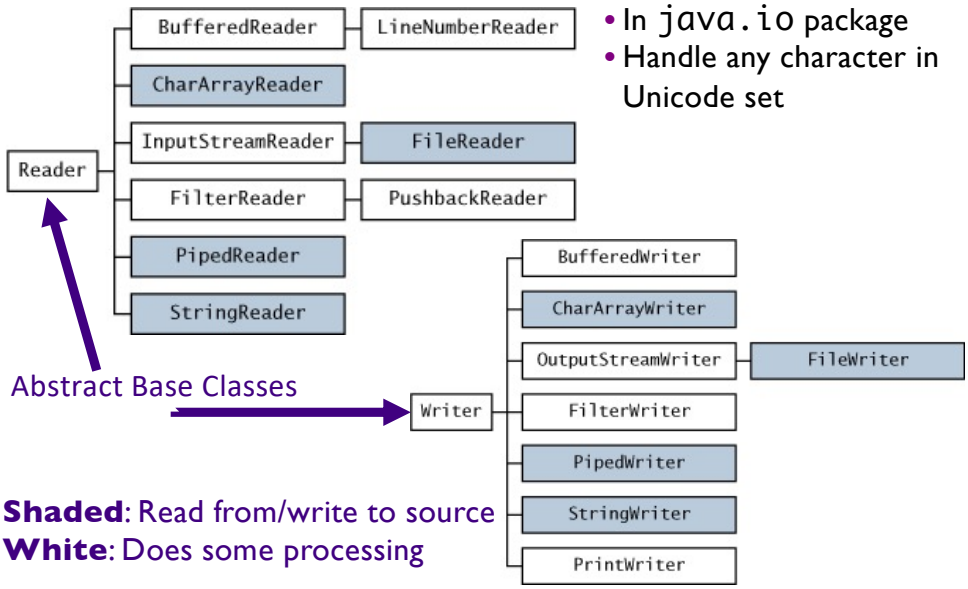


In `java.io` package

Abstract Base Classes

Shaded: Read from/write to source
White: Does some processing

Character Streams: For Text



- In `java.io` package
- Handle any character in Unicode set

Abstract Base Classes

Shaded: Read from/write to source
White: Does some processing

Reading Text from a Stream: BufferedReader

- There is no PrintReader class
- Constructor requires a Reader object

```
BufferedReader in = new BufferedReader( new FileReader("myfile.txt"));
```

- Read file, line-by-line using readLine()
  - Reads in a line of text and returns it as a String
  - Returns null when no more input is available

```
String line;
while ((line = in.readLine()) != null) {
        // process the line
}
```


Reading Text from a Stream

- You can attach a BufferedReader to an InputStreamReader:

```
BufferedReader consoleReader= new BufferedReader(
        new InputStreamReader(System.in));
BufferedReader webpageReader = new BufferedReader(
        new InputStreamReader(url.openStream());
```

Note how easy it is to read from different sources

- *Used* to be the best way to read from the console

Scanners

- Scanners do not throw `IOExceptions`!
  - For a simple console program, `main()` does not have to deal with or throw `IOExceptions`
  - Handling those [checked] exceptions is required with `BufferedReader/InputStreamReader` combination
- Throws `InputMismatchException` when token doesn't match pattern for expected type
  - e.g., `nextLong()` called with next token "AAA"
  - No catching required

Meaning it is what type of exception?
How do you prevent errors in Scanner?

Scanners

- Scanners do not throw `IOExceptions`!
  - For a simple console program, `main()` does not have to deal with or throw `IOExceptions`
  - Handling those [checked] exceptions is required with `BufferedReader/InputStreamReader` combination
- Throws `InputMismatchException` when token doesn't match pattern for expected type
  - e.g., `nextLong()` called with next token "AAA"
  - `RuntimeException` (no catching required)

How do you prevent such errors?

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

- 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

16

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

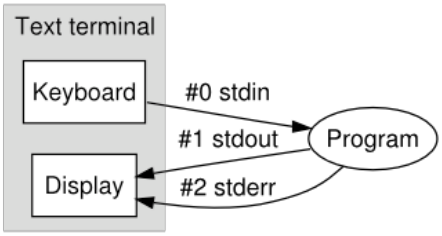
17

STANDARD ERROR

18

Standard Streams

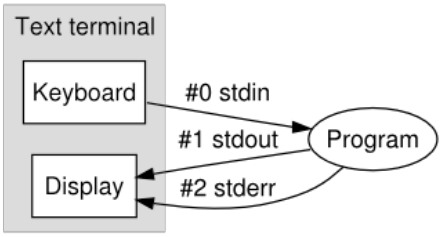
- Preconnected streams
  - Standard Out: stdout
  - Standard In: stdin
  - *Standard Error: stderr*
    - For error messages and diagnostics
    - In Java: `System.err`



Benefits of two output streams (out and err)?

19

Standard Streams



- Preconnected streams
  - Standard Out: stdout
  - Standard In: stdin
  - *Standard Error: stderr*
    - For error messages and diagnostics
    - In Java: `System.err`
- Helpful to separate *output* vs *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)

Standard Streams

- Documentation for Python's `print` function:

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

- `file` parameter says where to direct output
  - Default is to standard out

How could you print to standard error?

Standard Streams

- Documentation for Python's `print` function:

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

- `file` parameter says where to direct output

```
import sys
print("Hello!")
print("Error Hello!", file=sys.stderr)
```


Redirecting Output

- Recall earlier this semester

```
$ java Assign1 > debugged.out
```

  - Redirected `stdout` to `debugged.out`
  - `stderr` would still go to terminal

- To redirect `stderr` to same file as well

```
$ java Assign1 1> debugged.out 2>&1
```

`StandardStreamsExample.java`

JAR FILES

Jar (Java Archive) Files

- Archives of Java files
- Package code into a neat bundle to distribute
  - Easier, faster to download
  - Easier for others to use
- `jar` command: create, view, and extract Jar files
  - Works similarly to `tar`

```
jar cf myapplication.jar *.class
```

- Run it using `java`

```
java -jar myapplication.jar
```

Jar/Tar Commands

- Common options:

Option/ Operations	Meaning
f	The name of the archive **f**ile
c	**C**reate an archive file
x	E**x**tract the archive file
v	**V**erbose
z	**Z**ip (compress)
t	**T**able of contents (list contents)

- Common use:
  - `jar cfz code.jar.gz class_files_directory`
  - `jar xfz code.jar.gz`

Jar file: Metadata

- Jar file includes a special metadata file with the path `META-INF/MANIFEST.MF`
  - Say how Jar file is used
  - `jar` creates a default metadata file, if not specified

Jar file: Metadata

- Example metadata file that allows you to execute the JAR with java

```
Manifest-Version: 1.0
Main-Class: MyApplication

```

Note the newline

- To create the jar file:

Specifying the metadata file

```
jar cmf myManifest myapplication.jar *.class
```

- Run it using java

```
java -jar myapplication.jar
```


Creating Jar Files in Eclipse

- Export → Java → Jar file
  - Options to create a MANIFEST.MF file
  - Options to include source files or only class files

Typical Scenario with Jar Files

- "I want to use this third-party (not part of Java library) library in my code"
- You have a *jar* file of the code
- And then you add the jar file to your ***classpath***

30

CLASSPATH

31

Classpath

- Tells the compiler or JVM where to look for user-defined classes and packages (jar files)
  - Often when using third-party libraries
- Similar to PYTHONPATH
- Typically know it needs to be set when there are "Class not found" error messages in your code but you have the appropriate import

Setting the Classpath

- Can specify classpath in command line

```
javac -cp path/to/myjavaclasses MyClass.java
java –cp path/to/myjavaclasses MyClass
```

Can be .class files or jar files

- Can specify the classpath environment variable
  - Edit your `.bash_profile` OR
  - Set in terminal

```
CLASSPATH=$CLASSPATH:path/to/myjavaclasses
echo $CLASSPATH
```

Current value of CLASSPATH

- In Eclipse, you can "Configure Build Path" for a project

COMPILATION

34

Review

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

35

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

Summary:

In pure forms

Compiled vs Interpreted Languages

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

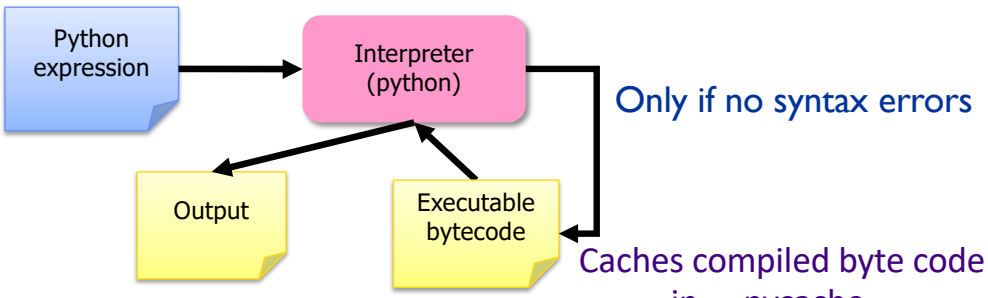
Interpreted

- ✓ 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

- 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 does it make the code more efficient?
- For each optimization, should you do these optimizations yourself? 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

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

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 Examples

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 Examples

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 Tradeoffs

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

Looking Ahead

- Tomorrow night: Assignment 6