Objectives

- Jar files
- Exceptions
  - Wrap up
  - Why Exceptions?
- Files
- Streams

JAR FILES

Jar (Java **Ar**chive) 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 archive.jar.gz arch_directory`
  - `jar xfz archive.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

```

- To create the jar 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
- Should submit assignments this way
  - ***Must include source files***
    - Look for checkbox

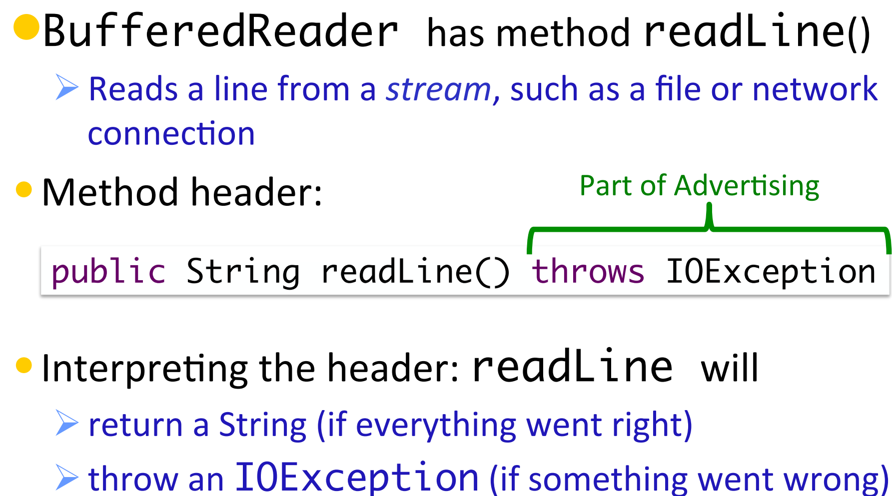
EXCEPTIONS

Discussion: Why Checked and Unchecked Exceptions?

- Why do we have exceptions that the compiler doesn't force the programmer to check?
  - Think about examples of unchecked exceptions (`ArrayOutOfBoundsException`, `NullPointerException`, `ClassCastException`) and when those exceptions can occur

Methods and Exceptions Example

- `BufferedReader` has method `readLine()`
  - Reads a line from a *stream*, such as a file or network connection
- Method header:




```
public String readLine() throws IOException
```

- Interpreting the header: `readLine` will
  - return a String (if everything went right)
  - throw an `IOException` (if something went wrong)

Advertising Checked Exceptions

- Advertising: in Javadoc, document under what conditions each exception is thrown
  - `@throws` tag
- Examples of when your method should advertise the ***checked*** exceptions that it may throw
  - Your method calls a method that throws a checked exception
  - Your method detects an error in its processing and decides to throw an exception

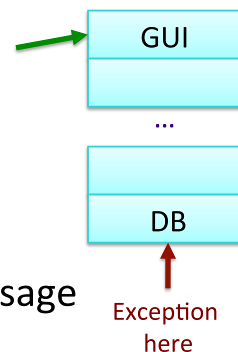
Javadoc Guidelines about @throws

- Always report if throw **checked** exceptions
- Report any unchecked exceptions that the caller might reasonably want to catch
  - Exception: `NullPointerException`
  - Allows caller to handle (or not)
  - Document exceptions that are independent of the underlying implementation
- Errors should **not** be documented as they are unpredictable

What to do with a Caught Exception?

- Dump the stack after the exception occurs
  - What else can we do?
- Generally, two options:
  1. Catch the exception and recover from it
  2. Pass exception up to whoever called it

To Throw or Catch?



- Problem: lower-level exception propagated up to higher-level code
- Example: user enters account information and gets exception message "field exceeds allowed length in database"
  - Lost context
  - Lower-level detail polluting higher-level API

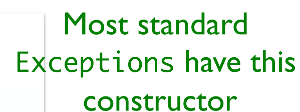
Solution: higher-levels should catch lower-level exceptions and throw them in terms of higher-level abstraction

Exception Translation

```
try {
    // Call lower-level abstraction
}
catch (LowerLevelException ex) {
    // log exception …
    throw new HigherLevelException(…);
}
```

- Special case: Exception Chaining
  - When higher-level exception needs info from lower-level exception

```
try {
    // Call lower-level abstraction
}
catch (LowerLevelException cause) {
    // log exception …
    throw new HigherLevelException(cause);
}
```



Summary: Methods Throwing Exceptions

- API documentation tells you if a method can throw an exception
  - If so, you **must** handle it
- If your method could possibly throw an exception (by generating it or by calling another method that could), advertise it!
  - If you can't handle every error, that's OK…let whoever is calling you worry about it
  - However, they can only handle the error if you advertise the exceptions you can't deal with

Programming with Exceptions

- Exception handling is slow
- Use one big `try` block instead of nesting `try-catch` blocks
  - Speeds up Exception Handling
  - Otherwise, code gets too messy
- Don't ignore exceptions (e.g., `catch` block does nothing)
  - Better to pass them along to higher calls

```
try {
}
catch () {
}
try {
}
catch () {
}
```

```
try {
    try {
    }
    catch () {
    }
}
catch () {
}
```

```
try {
    …
    …
}
catch () {
}
```


Creating Our Own Exception Class

- Try to reuse an existing exception
  - Match in name as well as semantics
- If you cannot find a predefined Java `Exception` class that describes your condition, implement a new `Exception` class!

Creating Our Own Exception Class

```
public class FileFormatException extends IOException {
      public FileFormatException() {

      }

      public FileFormatException(String message) {
            super(message);
      }

      // other 2 standard constructors…
}
```

What happens in this constructor implicitly?

Is this a checked or unchecked exception?

- Can now throw exceptions of type **`FileFormatException`**

Guidelines for Creating Your Own Exception Classes

- Include accessor methods to get more information about the cause of the exception
  - "failure-capture information"
- Checked or unchecked exception?
  - Checked: *forces* API user to handle BUT more difficult to use API
    - Has to handle all checked exceptions
  - Use checked exception if exceptional condition cannot be prevented by proper use of API *and* API user can take a useful action afterward

Discussion: Benefits of Exceptions

- Been talking about details…
- Why does Java have exceptions as part of the language?
- Why does Java add some features that Python doesn't have?

Benefits of Exceptions

- Force error checking/handling
  - Otherwise, won't compile
  - Does not guarantee "good" exception handling
- Ease debugging
  - Stack trace
- Separates error-handling code from "regular" code
  - Error code is in catch blocks at end
  - Descriptive messages with exceptions
- Propagate methods up call stack
  - Let whoever "cares" about error handle it
- Group and differentiate error types

FILES

`java.io.File` Class

- Represents a file or directory
- Provides functionality such as
  - Storage of the file on the disk
  - Determine if a particular file exists
  - When file was last modified
  - Rename file
  - Remove/delete file
  - …

Making a File Object

- Simplest constructor takes full file name (including path)
  - If don't supply path, Java assumes current directory (.)

```
File f1 = new File("chicken.data");
```

  - Creates a `File` *object* representing a file named "chicken.data" in the current directory
  - Does ***not*** create a file with this name on disk

Files, Directories, and Useful Methods

- A `File` object can represent a file **or** a directory
  - Directories are special files in most modern operating systems
- Use `isDirectory()` and/or `isFile()` for type of file `File` object represents
- Use `exists()` method
  - Determines if a file exists on the disk

More File Constructors

- String for the path, String for filename

```
File f2 = new File(
      "/csdept/local/courses/cs209/handouts",
      "chicken.data");
```

- File for directory, String for filename

```
File dir = new File(
      "/csdept/local/courses/cs209/handouts");
File f4 = new File(dir, "chicken.data");
```


“Break” any of Java’s Principles?

`java.io.File` Class

- 25+ methods
  - Manipulate files and directories
  - Creating and removing directories
  - Making, renaming, and deleting files
  - Information about file (size, last modified)
  - Creating temporary files
  - …
- See online API documentation

`FileTest.java`

java.util.

SCANNER

java.util.Scanner

- New(er) class for handling input
  - Since Java 1.5
- Many constructors
  - Read from file, input stream, string …

```
Scanner sc = new Scanner(System.in);
```

- Many methods
  - nextXXXX (int, long, line)
  - Skipping patterns, matching patterns, etc.

Scanners

- Breaks its input into tokens using a delimiter pattern, which matches whitespace

What is "delimiter pattern"?
What is "whitespace"?

- Converts resulting tokens into values of different types using nextXXX()
- Can change token delimiter from default of whitespace
- Assumes numbers are input as decimal
  - Can specify a different radix

Using Scanners

- Use *nextXXX()* to read from it…

```
long tempLong;

// create the scanner for the console
Scanner sc = new Scanner(System.in);

// read in an integer and a String
int i = sc.nextInt();
String restOfLine = sc.nextLine();

// read in a bunch of long integers
while (sc.hasNextLong()) {
      tempLong = sc.nextLong();
}
```


Using Scanner

Simplified version of online example

```
public static void main(String[] args) {

    // open the Scanner on the console input, System.in
    Scanner scan = new Scanner(System.in);
    scan.useDelimiter("\n"); // breaks up by lines, useful for
                             // console I/O

    System.out.print("Please enter the width of a rectangle: ");
    int width = scan.nextInt();

    System.out.print("Please enter the height of a rectangle: ");
    int length = scan.nextInt();

    System.out
       .println("The area of your square is " + length * width +
                    ".");
}
```

ConsoleUsingScannerDemo.java

Output

Read in as one token

```
This program calculates the area of a rectangle.

Please enter the width of a rectangle (as an integer):
the number is 1
Incorrect input.
Please enter the width of a rectangle (as an integer):
1 2
Incorrect input.
Please enter the width of a rectangle (as an integer):
2
Please enter the height of a rectangle (as an
integer): 3
The area of your rectangle is 6.
```


Scanners & Exceptions

- Scanners do not throw `IOExceptions`!
  - For a simple console program, `main()` does not have to deal with or throw `IOExceptions`
  - 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?

`Console` class

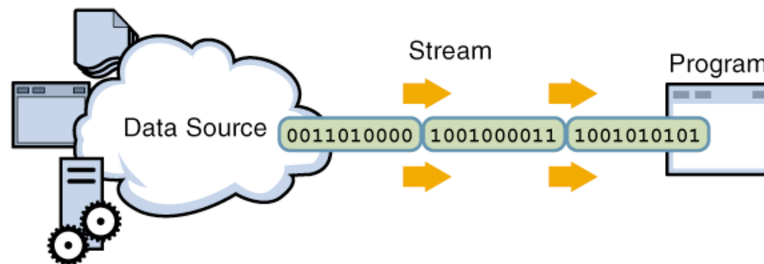
- Get a `Console` object using `System.console()`
- Has some useful methods for requesting passwords
- Issue: does not work through an IDE

`ConsoleUsingConsoleDemo.java`

STREAMS

Streams

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

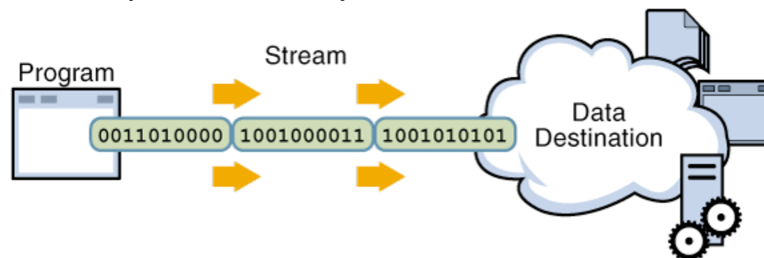


input stream: an object from which we can ***read*** a sequence of bytes

abstract class: `java.io.InputStream`

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`

Java Streams

- MANY (80+) types of Java streams
- In `java.io` package
- Why **stream** abstraction?
  - Information stored in different sources is accessed in essentially the same way
    - Example sources: file, on a web server across the network, string
  - Allows same methods to read or write data, regardless of its source
    - Create an `InputStream` or `OutputStream` of the appropriate type

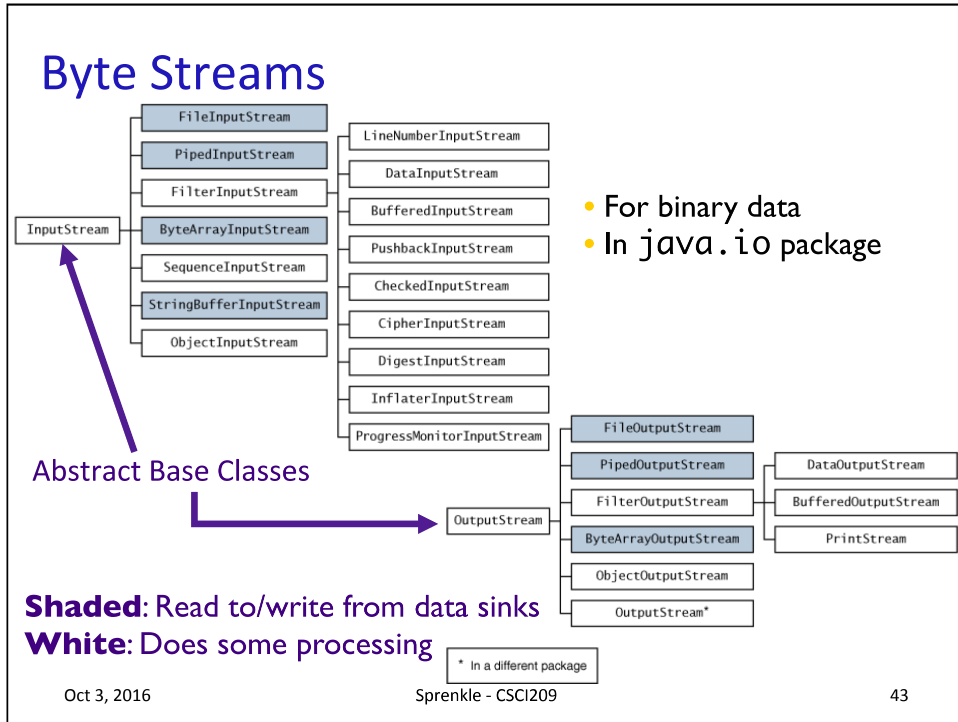
`java.io` Classes Overview

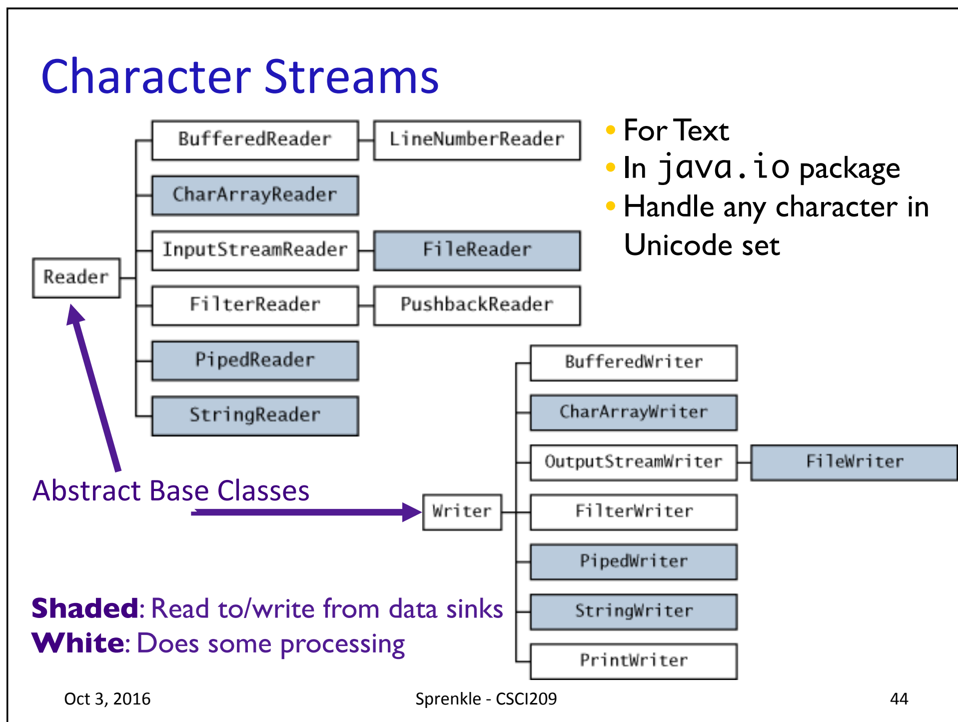
- Two types of stream classes, based on datatype: Byte, Text
- Abstract base classes for **binary** data:

  `InputStream` `OutputStream`

- Abstract base classes for **text** data:

  `Reader` `Writer`


Byte Streams
FileInputStream
PipedInputStream
FilterInputStream
InputStream
ByteArrayInputStream
SequenceInputStream
StringBufferInputStream
ObjectInputStream
LineNumberInputStream
DataInputStream
BufferedInputStream
PushbackInputStream
CheckedInputStream
CipherInputStream
DigestInputStream
InflaterInputStream
ProgressMonitorInputStream
For binary data
In java.io package
Abstract Base Classes
OutputStream
FileOutputStream
PipedOutputStream
FilterOutputStream
ByteArrayOutputStream
ObjectOutputStream
OutputStream*
DataOutputStream
BufferedOutputStream
PrintStream
Shaded: Read to/write from data sinks
White: Does some processing
* In a different package
Oct 3, 2016
Sprenkle - CSCI209
43


Character Streams
BufferedReader
LineNumberReader
CharArrayReader
InputStreamReader
FileReader
Reader
FilterReader
PushbackReader
PipedReader
StringReader
For Text
In java.io package
Handle any character in Unicode set
Abstract Base Classes
Writer
BufferedWriter
CharArrayWriter
OutputStreamWriter
FileWriter
FilterWriter
PipedWriter
StringWriter
PrintWriter
Shaded: Read to/write from data sinks
White: Does some processing
Oct 3, 2016
Sprenkle - CSCI209
44

Console I/O

- Output:
  - `System.out` is a `PrintStream` object
- Input
  - `System.in` is an `InputStream` object
  - Throws exceptions if format of input data is not correct
    - Handle in `try/catch`

Opening & Closing Streams

- Streams are *automatically opened* when constructed
- Close a stream by calling its `close()` method
  - Close a stream as soon as object is done with it
  - Free up system resources

Reading & Writing Bytes

- Abstract parent class: `InputStream`
  - `abstract int read()`
    - reads one byte from the stream and returns it
- Concrete input stream classes override `read()` to provide appropriate functionality
  - e.g., `FileInputStream`'s `read()` reads one byte from a file
- Similarly, `OutputStream` class has abstract `write()` to write a byte to the stream

Oct 3, 2016 Sprenkle - CSCI209 47

Reading & Writing Bytes

- `read()` and `write()` are ***blocking*** operations
  - If a byte cannot be read from the stream, the method waits (does not return) until a byte is read
- `available()` : get the number of bytes that are available for reading
- Example use:

```
int bytesAvailable = System.in.available();
if (bytesAvailable > 0)
     System.in.read(byteBuffer);
```

Oct 3, 2016 Sprenkle - CSCI209 48