

Objectives

- Exceptions
- Files
- Streams

1

Review

1. What is an Exception?
2. How do we create Exceptions?
3. How do we *advertise* that our method may produce an exception?
4. What are the different categories of exceptions?
 - a) What are examples (i.e., class names) of those categories of exceptions?
5. How do you *handle* an exception? (In Python, this was called “except”)
6. How do we make a block of code execute regardless of whether some code threw an exception or not?
7. What is Eclipse? What can it do?
 - a) Why did I wait until now to show you Eclipse?

2

Eclipse Tradeoffs

- Very helpful – *after* you know what you’re doing
 - You know
 - Code is compiled before executed
 - Structure of classes
 - How to fix errors
- Eclipse can handle the “routine” for you
 - That wasn’t “routine” for you a few weeks ago
 - Help you focus on the important design considerations
- Gives suggestions for fixes
 - You need to think through what the appropriate fix is
 - Sometimes, it’s “I’m not done yet”
 - Don’t say “Eclipse made me do <something>”
- Eclipse is a beast (memory hog)
 - If you have less than ~8GB of memory, Eclipse will be slow

Oct 13, 2021

Sprenkle - CSCI209

3

3

Eclipse Hints

- After you have written a method, type


```
/**
```

before the method, and then hit enter and the Javadocs comment template will be automatically generated for you

 - Use (command/control)-spacebar for possible completions
 - Use (command/control)-shift-F to format code

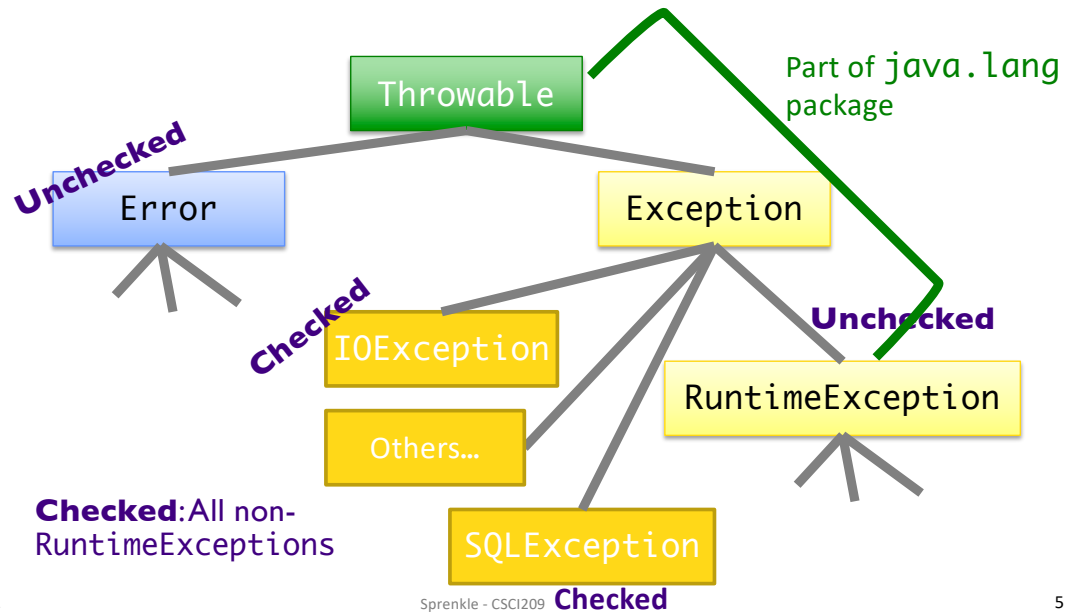
Oct 13, 2021

Sprenkle - CSCI209

4

4

Review: Exception Classification



5

Review: Categories of Exceptions

Unchecked

- Any exception that derives from `Error` or `RuntimeException`
- Programmer does not necessarily create/handle
- Try to prevent them
- Often indicates programmer error
 - E.g., precondition violations, not using API correctly

Checked

- Any other exception
- Programmer creates and handles checked exceptions
- Compiler-enforced checking
 - Improves *reliability**
- For conditions from which caller can reasonably be expected to recover

Oct 13, 2021

Sprenkle - CSCI209

6

6

Review: Types of Unchecked Exceptions

1. Derived from the class Error

- Any line of code can generate because it is an internal JVM error
- Don't worry about what to do if this happens

2. Derived from the class RuntimeException

- Indicates a bug in the program
- Fix the bug
- Examples: `ArrayOutOfBoundsException`, `NullPointerException`, `ClassCastException`

Oct 13, 2021

Sprenkle - CSCI209

7

7

Review: Throwing An Exception We Created

1. Create a new object of class `IllegalArgumentException`

- Class derived from `RuntimeException`

2. `throw` it

- Method ends at this point
- Calling method handles exception

```
if (grade < 0 || grade > 100) {
    throw new IllegalArgumentException(
        "Grade is not in valid range (0-100)");
}
```

8

Review: try/catch/finally Blocks

```
try {
    statement1;
    statement2;
}
catch (EOFException e) {
    statement3;
    statement4;
}
finally {
    statement5;
}
```

- Which statements run if:
 1. Neither statement1 nor statement2 throws an exception
 - 1, 2, 5
 2. statement1 throws an EOFException
 - 1,3,4,5
 3. statement2 throws an EOFException
 - 1,2,3,4,5
 4. statement1 throws an IOException
 - 1,5

Oct 13, 2021

Sprenkle - CSCI209

9

9

What to do with a Caught Exception?

- Print the stack after the exception occurs
 - For now, printed to console; better: a log file
 - What else can we do?
- Generally, two options:
 1. Catch the exception and recover from it
 - Recovery: reset state; clean up
 2. Pass exception up to whoever called it

Oct 13, 2021

Sprenkle - CSCI209

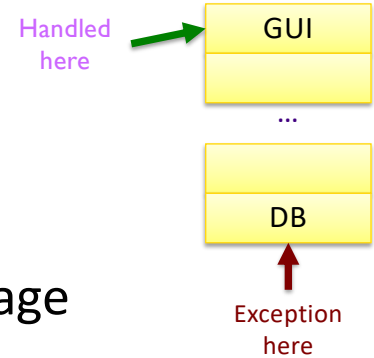
10

10

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



Oct 13, 2021

Sprenkle - CSCI209

11

11

Exception Translation

- Special case:
Exception Chaining

- When higher-level exception needs info from lower-level exception

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

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

Most standard
Exceptions have this
constructor

Oct 13, 2021

Sprenkle - CSCI209

12

12

Guidelines for Exception Translation

- Try to ensure that lower-level APIs succeed
 - Ex: verify that your parameters satisfy invariants
- Insulate higher-level from lower-level exceptions
 - Handle in some reasonable way
 - Always log problem so admin can check
- If can't do previous two, then use exception translation

Oct 13, 2021

Sprenkle - CSCI209

13

13

Programming with Exceptions

- Exception handling is slow
- Group relevant code together
 - Scope of try/catch block should be small
- 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 () {
}
```

Oct 13, 2021

Sprenkle - CSCI209

14

14

Programming with Exceptions

- Typical Scenario: Your code calls a method that throws a checked exception
 - Code will not compile until you *handle* the exception
- You have 2 options:
 - 1) Add throws declaration
 - 2) Surround with try/catch

Oct 13, 2021

Sprenkle - CSCI209

Demo in Eclipse

15

15

try Block Scope Example

```

public void read(BufferedReader in) {
    try {
        boolean done = false;
        while (!done) {
            String line=in.readLine();
            // above could throw IOException
            if (line == null)
                done = true;
        }
    }
    catch {
    }
}

```

Only this line can throw the exception.

But all of this code is in **try** block
Why? My considerations:

- In while loop
- Scope of variables
- Readability of code

Oct 13, 2021

Sprenkle - CSCI209

16

16

Try/Catch Block Example Alternatives

```
public void read(BufferedReader in) {
    boolean done = false;
    try {
        while (!done) {
            String line = in.readLine();
            if (line == null)
                done = true;
        }
    }
    catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

Move done initialization outside try

```
public void read(BufferedReader in) {
    boolean done = false;
    while (!done) {
        try {
            String line = in.readLine();
            if (line == null)
                done = true;
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Put try closer to possible offender

Oct 13, 2021

Sprenkle - CSCI209

17

17

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
 - They can only handle if you advertise the exceptions you can't deal with

Oct 13, 2021

Sprenkle - CSCI209

18

18

Creating Custom Exception Class

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

Skipping next 3 slides in class

Discussion: Benefits of Exceptions

- Been talking about details...
- Why does Java have exceptions as part of the language?

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 errors up call stack
 - Let whoever "cares" about error handle it
- Group and differentiate error types

Does **NOT** mean that error is prevented at compile time—just that we can improve robustness

Oct 13, 2021

Sprenkle - CSCI209

24

24

FILES

Oct 13, 2021

Sprenkle - CSCI209

25

25

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
 - ...

Oct 13, 2021

Sprenkle - CSCI209

26

26

Making a File Object

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


```
File myFile = 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
- Similar to Python:

```
myFile = open("chicken.data")
```

Oct 13, 2021

Sprenkle - CSCI209

27

27

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

In Python, functionality are in the `os.path` module

Oct 13, 2021

Sprenkle - CSCI209

28

28

More File Constructors

- String for the path, String for filename

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

- File for directory, String for filename

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

Oct 13, 2021

Sprenkle - CSCI209

29

29

“Break” any of Java’s Principles?

Oct 13, 2021

Sprenkle - CSCI209

30

30

“Break” any of Java’s Principles?

- Principle of Portability
 - Write and Compile Once, Run Anywhere
- Problem: file paths are OS-specific
- `java.io.File.separator`
 - OSX/Linux: `/`
 - Windows: `\`
- Takeaways:
 - Use relative paths
 - Use configuration files to set paths

Oct 13, 2021

Sprenkle - CSCI209

31

31

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

Oct 13, 2021

Sprenkle - CSCI209

FileTest.java

32

32

STREAMS

Oct 13, 2021

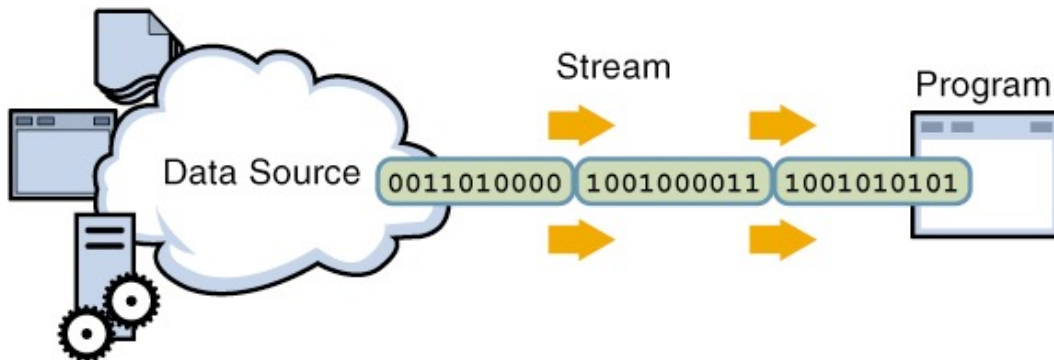
Sprenkle - CSCI209

33

33

Streams

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

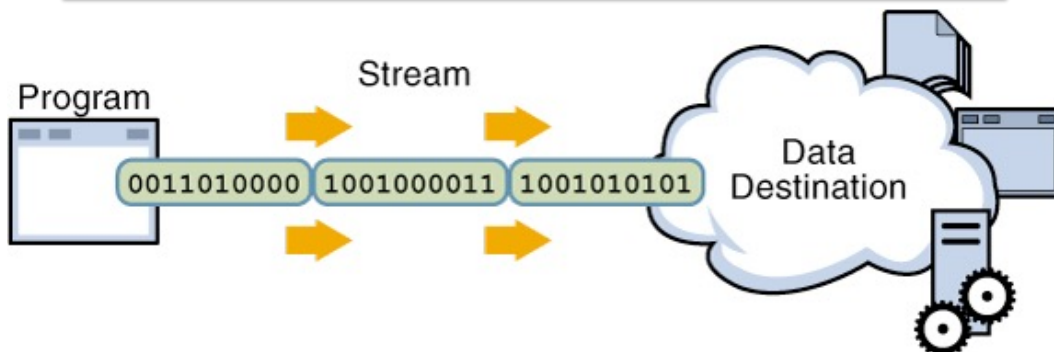


input stream: an object from which we can **read** a **sequence** of bytes
abstract class: `java.io.InputStream`

34

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`

Oct 13, 2021

Sprenkle - CSCI209

35

35

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

Oct 13, 2021

Sprenkle - CSCI209

36

36

`java.io` Classes Overview

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

InputStream

OutputStream

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

Reader

Writer

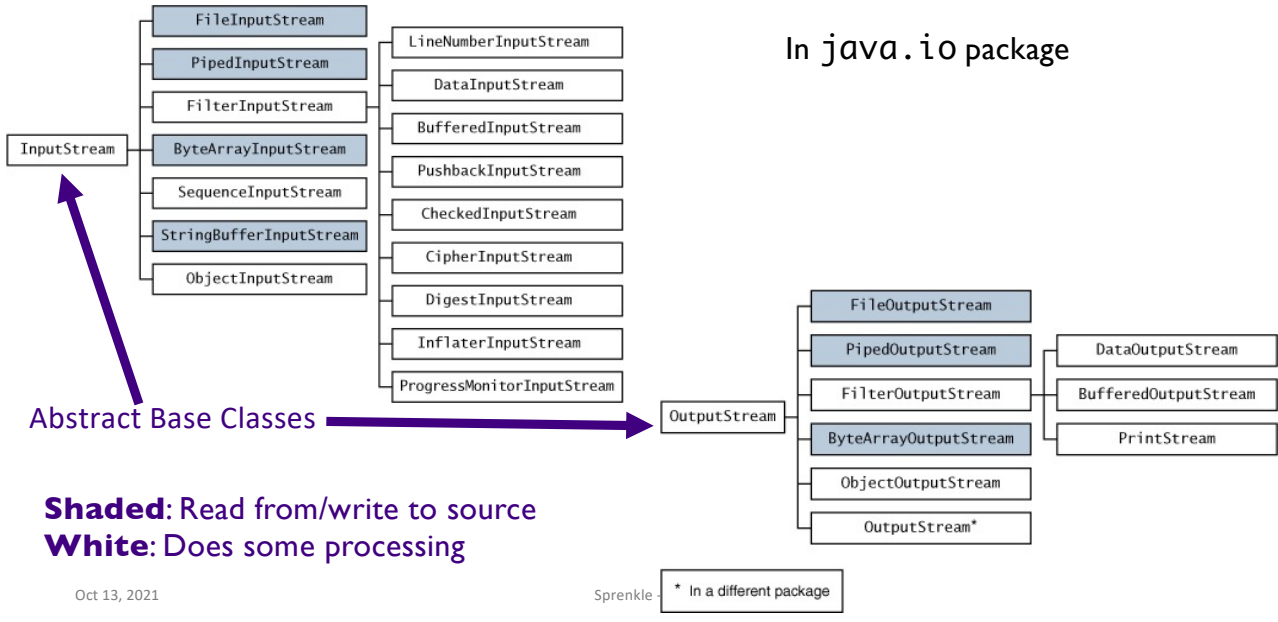
Oct 13, 2021

Sprenkle - CSCI209

37

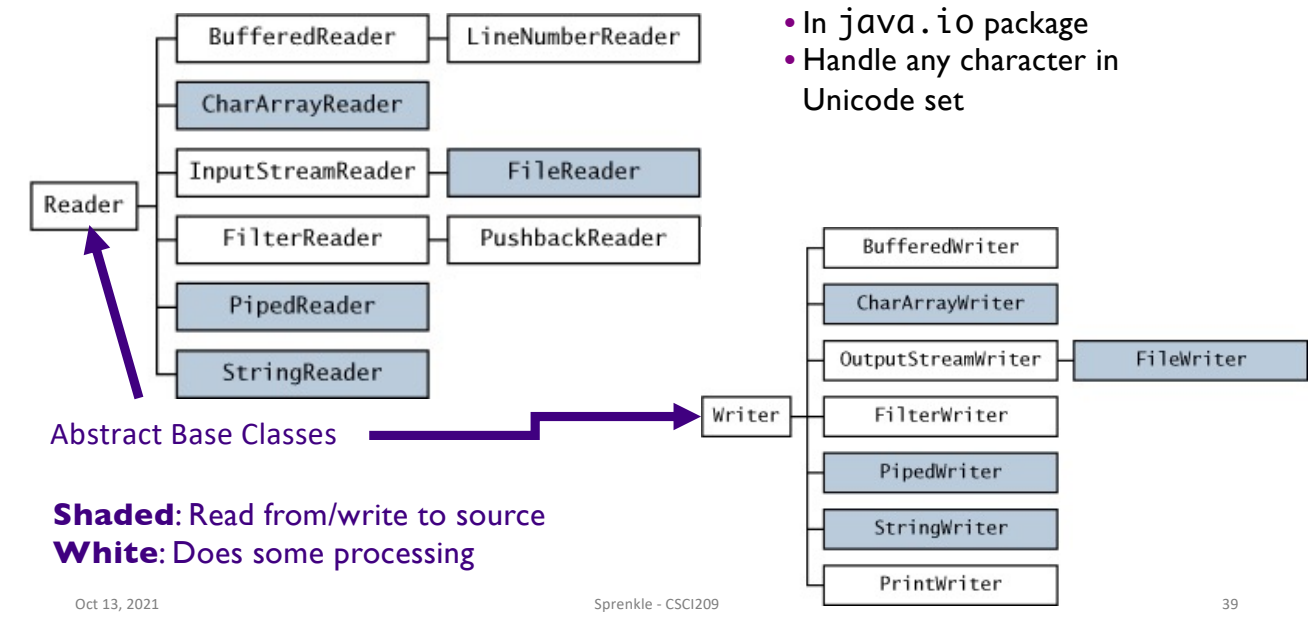
37

Byte Streams: For Binary Data



38

Character Streams: For Text



39

Console I/O

- Output:
 - `System.out` is a `PrintStream` object
- Input
 - `System.in` is an `InputStream` object
 - Throws exceptions if errors when reading
 - Handle in `try/catch`

`SystemIOStarter.java`

Oct 13, 2021

Sprenkle - CSCI209

40

40

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

Oct 13, 2021

Sprenkle - CSCI209

41

41

Reading & Writing Bytes

- Abstract parent class: **InputStream**
 - **abstract int read()**
 - reads one byte from the stream and returns it
 - Concrete child 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 13, 2021

Sprenkle - CSCI209

42

42

File Input and Output Streams

- **FileInputStream**: provides an input stream that can read from a file

- Constructor takes the name of the file:

```
FileInputStream fin = new FileInputStream("chicken.data");
```

- Or, uses a **File** object ...

```
File inputFile = new File("chicken.data");
FileInputStream fin = new FileInputStream(inputFile);
```

Oct 13, 2021

Sprenkle - CSCI209

FileTest.java

44

44

More Powerful Stream Objects

- **DataInputStream**

- Reads Java primitive types through methods such as `readDouble()`, `readChar()`, `readBoolean()`

- **DataOutputStream**

- Writes Java primitive types with `writeDouble()`, `writeChar()`, `writeBoolean()`, ...

Oct 13, 2021

Sprenkle - CSCI209

45

45

Connected Streams

Our goal: read numbers from a file

- `FileInputStream` can read from a file but has no methods to read numeric types
- `DataInputStream` can read numeric types but has no methods to read from a file
- Java allows you to **combine** two types of streams into a **connected stream**
 - `FileInputStream` → chocolate
 - `DataInputStream` → peanut butter

Oct 13, 2021

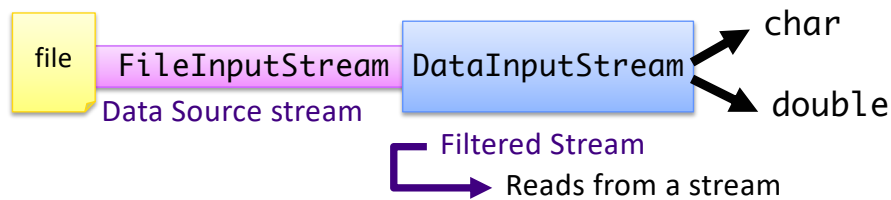
Sprenkle - CSCI209

46

46

Connected Streams

- Think of a stream as a pipe
- `FileInputStream` knows how to read from a file
- `DataInputStream` knows how to read an `InputStream` into useful types
- Connect **out** end of `FileInputStream` to **in** end of `DataInputStream`...



Oct 13, 2021

Sprenkle - CSCI209

47

47

Connecting Streams

- If we want to read numbers from a file
 - `FileInputStream` reads bytes from file
 - `DataInputStream` handles numeric type reading
- Connect the `DataInputStream` to the `FileInputStream`
 - `FileInputStream` gets the bytes from the file and `DataInputStream` reads them as assembled types

```
FileInputStream fin = new FileInputStream("chicken.data");
DataInputStream din = new DataInputStream(fin);
double num1 = din.readDouble();
```

"wrap" fin in din

Oct 13, 2021

Sprenkle - CSCI209

DataIODemo.java

48

48

Data Source vs. Filtered Streams

Data Source Streams

- Communicate with a data source
 - file, byte array, network socket, or URL

Filtered Streams

- Subclasses of `FilterInputStream` or `FilterOutputStream`
- Always contains/connects to another stream
- Adds functionality to other stream
 - Automatically buffered IO
 - Automatic compression
 - Automatic encryption
 - Automatic conversion between objects and bytes

Oct 13, 2021

Sprenkle - CSCI209

49

49

Another Filtered Stream: Buffered Streams

• `BufferedInputStream` buffers your input streams

- A pipe in the chain that adds *buffering* → speeds up access

```
DataInputStream din = new DataInputStream (
    new BufferedInputStream (
        new FileInputStream("chicken.data")));
```



Review: What functionality does each stream add?

Oct 13, 2021

50

50

Connected Streams: Similar for Output

- Example: for buffered output to the file and to write types
 - Create a `FileOutputStream`
 - Attach a `BufferedOutputStream`
 - Attach a `DataOutputStream`
 - Perform typed writing using methods of the `DataOutputStream` object

Combine different types of streams
to get functionality you want

Oct 13, 2021

Sprenkle - CSCI209

51

51

TEXT STREAMS

Oct 13, 2021

Sprenkle - CSCI209

52

52

Text Streams

- Previous streams: operate on *binary* data, not text
- Java uses Unicode to represent characters/strings and some operating systems do not
 - Need something that converts characters from Unicode to whatever encoding the underlying operating system uses
 - Luckily, this is mostly hidden from you

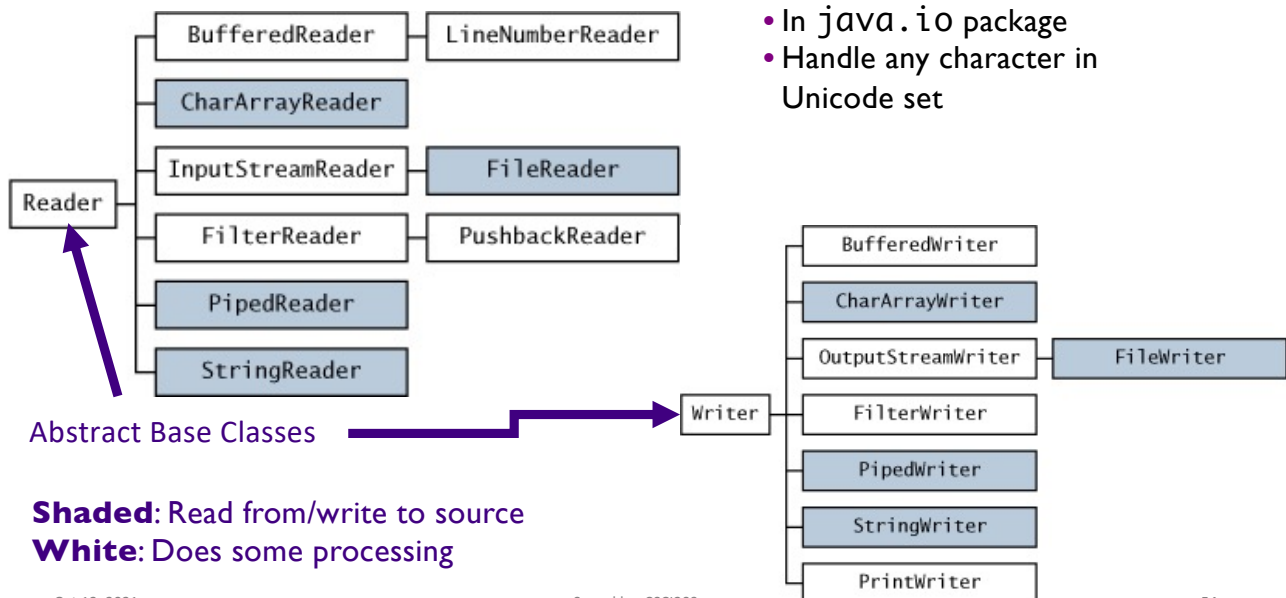
Oct 13, 2021

Sprenkle - CSCI209

53

53

Character Streams: For Text



Oct 13, 2021

Sprenkle - CSCI209

54

54

Text Streams

- Derived from **Reader** and **Writer** classes
 - Reader and Writer generally refer to **text I/O**
- Example: Make an input reader of type **InputStreamReader** that reads from keyboard

```
InputStreamReader in = new InputStreamReader(System.in);
```

- **in** reads characters from keyboard and converts them into Unicode for Java

Oct 13, 2021

Sprenkle - CSCI209

55


55

Text Streams and Encodings

- Attach an **InputStreamReader** to a **FileInputStream**
 - Assumes file has been encoded in the default encoding of underlying OS
- Can specify a different *encoding* in constructor of **InputStreamReader**

```
InputStreamReader in = new InputStreamReader(
    new FileInputStream("employee.data"));
```

```
InputStreamReader in = new InputStreamReader(
    new FileInputStream("employee.data"), "UTF-8");
```



Oct 13, 2021

56

56

Convenience Classes

- Reading and writing to text files is common
- **FileReader**
 - Convenience class *combines* a `InputStreamReader` with a `FileInputStream`
- Similar for output to text file

```
FileWriter out = new FileWriter("output.txt");
```

is equivalent to

```
OutputStreamWriter out = new OutputStreamWriter(
    new FileOutputStream("output.txt"));
```

Oct 13, 2021

Sprenkle - CSCI209

57

57

PrintWriter

- Easiest writer to use for writing text output
- Has methods for printing various data types
 - similar to a `DataOutputStream`, `PrintStream`
- Methods: `print`, `printf` and `println`
 - Similar to `System.out` (a `PrintStream`) to display strings

Oct 13, 2021

Sprenkle - CSCI209

58

58

PrintWriter Example

File to write to

```
PrintWriter out = new PrintWriter("output.txt");

String myName = "Homer Simpson";
double mySalary = 35700;

out.print(myName);
out.print(" makes ");
out.print(salary);
out.println(" per year.");
or
out.println(myName + " makes " + salary +
            " per year.");
```

Oct 13, 2021

Sprenkle - CSCI209

59

59

Review: Formatted Output

- printf or format

```
double f1=3.14159, f2=1.45, total=9.43;
// simple formatting...
System.out.printf("%6.5f and %5.2f", f1, f2);
// getting fancy (%n = \n or \r\n)...
System.out.printf("%-6s%5.2f\n", "Tax:", total);
```

Oct 13, 2021

Sprenkle - CSCI209

60

60

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
}
```

Oct 13, 2021

62

62

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

Oct 13, 2021

Sprenkle - CSCI209

63

63

Looking Ahead: Assignment 6

- Eclipse practice
- Javadocs
 - See what the web pages look like from your comments!