

Objectives

- Finish up Exceptions
- Files
- Streams

Sept 30, 2011

Sprenkle - CSCI209

1

Review

- What are the two types of *exceptions*?
- What are some ways to handle exceptions?
- What does it mean to “advertise” an exception?

Sept 30, 2011

Sprenkle - CSCI209

2

Practice: try/catch/finally Blocks

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

- Which statements run if:
 - Neither statement1 nor statement2 throws an exception
 - statement1 throws an EOFException
 - statement2 throws an EOFException
 - statement1 throws an IOException

Sept 30, 2011

Sprenkle - CSCI209

3

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

Sept 30, 2011

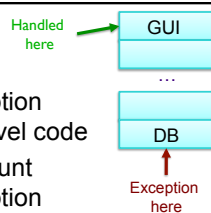
Sprenkle - CSCI209

4

Design Decision: 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”

What do you think happened?
Is this a good solution?



Sept 30, 2011

Sprenkle - CSCI209

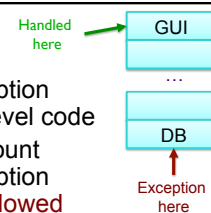
5

Design Decision: 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



Sept 30, 2011

Sprenkle - CSCI209

6

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);
}
```

Most standard Exceptions have this constructor

7

Guidelines for Exception Translation

- Avoidance!
 - 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

Sept 30, 2011

Sprenkle - CSCI209

8

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

Sept 30, 2011

Sprenkle - CSCI209

9

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 {
    try {
        catch O {
        }
    }
    catch O {
    }
}
try {
    -
    catch O {
    }
```

Sept 30, 2011

Sprenkle - CSCI209

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!

Sept 30, 2011

Sprenkle - CSCI209

11

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**

Sept 30, 2011

Sprenkle - CSCI209

12

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

Sept 30, 2011

Sprenkle - CSCI209

13

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?

Sept 30, 2011

Sprenkle - CSCI209

14

Benefits of Exceptions

- 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
- Checked exceptions: Force error checking/handling
 - Otherwise, won't compile
 - Does not guarantee “good” exception handling

Sept 30, 2011

Sprenkle - CSCI209

15

FILES

Sept 30, 2011

Sprenkle - CSCI209

16

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
 - ...

Sept 30, 2011

Sprenkle - CSCI209

17

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

Sept 30, 2011

Sprenkle - CSCI209

18

## 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

Sept 30, 2011

Sprenkle - CSCI209

19

## More File Constructors

- String for the path, String for filename

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

- File for directory, String for filename

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

Sept 30, 2011

Sprenkle - CSCI209

20

## "Break" any of Java's Principles?

Sept 30, 2011

Sprenkle - CSCI209

21

## Not Portable

- Accessing the file system is inherently not portable
  - In Windows, paths are "c:\\dir"
  - In Unix, paths are "/home/courses/dir"
- Relies on underlying file system/operating system to perform actions

Sept 30, 2011

Sprenkle - CSCI209

22

## Handling Portability Issues

- Static fields in `File` class
  - `static separator`
    - Unix: "/"
    - Windows: "\\"
  - `static pathSeparator`
    - For separating a list of paths
    - Unix: ":"
    - Windows: ";"
- Use relative paths, with separators

Why two \\?

Sept 30, 2011

Sprenkle - CSCI209

23

## 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

Sept 30, 2011

Sprenkle - CSCI209

24

## STREAMS

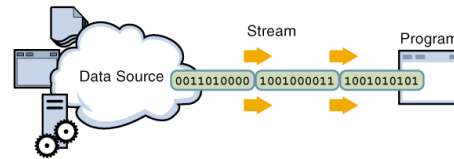
Sept 30, 2011

Sprenkle - CSCI209

25

## 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`

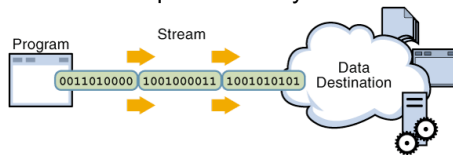
Sept 30, 2011

Sprenkle - CSCI209

26

## 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`

Sept 30, 2011

Sprenkle - CSCI209

27

## 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

Sept 30, 2011

Sprenkle - CSCI209

28

## java.io Classes Overview

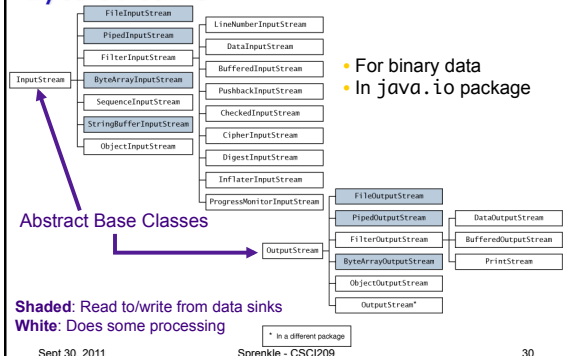
- Two types of stream classes, based on datatype: Byte, Text
- Abstract base classes for binary data:
  - `InputStream`
  - `OutputStream`
- Abstract base classes for text data:
  - `Reader`
  - `Writer`

Sept 30, 2011

Sprenkle - CSCI209

29

## Byte Streams

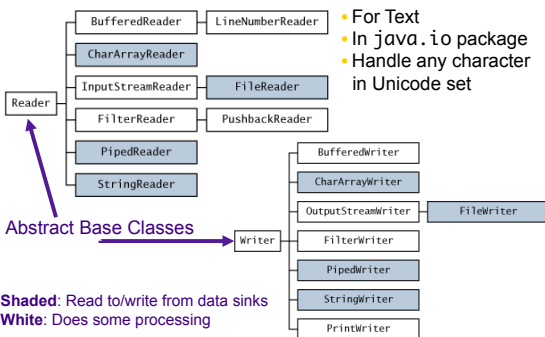


Sept 30, 2011

Sprenkle - CSCI209

30

## Character Streams



Sept 30, 2011

Sprenkle - CSCI209

31

## 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`

Sept 30, 2011

Sprenkle - CSCI209

32

## 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

Sept 30, 2011

Sprenkle - CSCI209

33

## 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

Sept 30, 2011

Sprenkle - CSCI209

34

## 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);
```

Sept 30, 2011

Sprenkle - CSCI209

35

## 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);
```

Sept 30, 2011

Sprenkle - CSCI209

FileTest2.java 36

## To Do

- Assignment 7
  - Modifying Olympic Score generator
    - Read difficulty score from console
    - Read execution scores from a file
      - Filename comes from console
  - Due Friday, but should start before midterm
- Next Wednesday: Midterm
  - Prep document online