

→

→



THE LECTURE

- **Recap**
- **Quick Quiz**
- **Object-Oriented – Inheritance**
 - Extending classes
 - Abstract classes
 - Interfaces
- **Design Patterns: Singleton**
- **File class**
- **Logging**

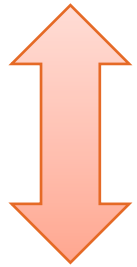
RECAP

MOTIVATIONS FOR OBJECT-ORIENTED DEVELOPMENT

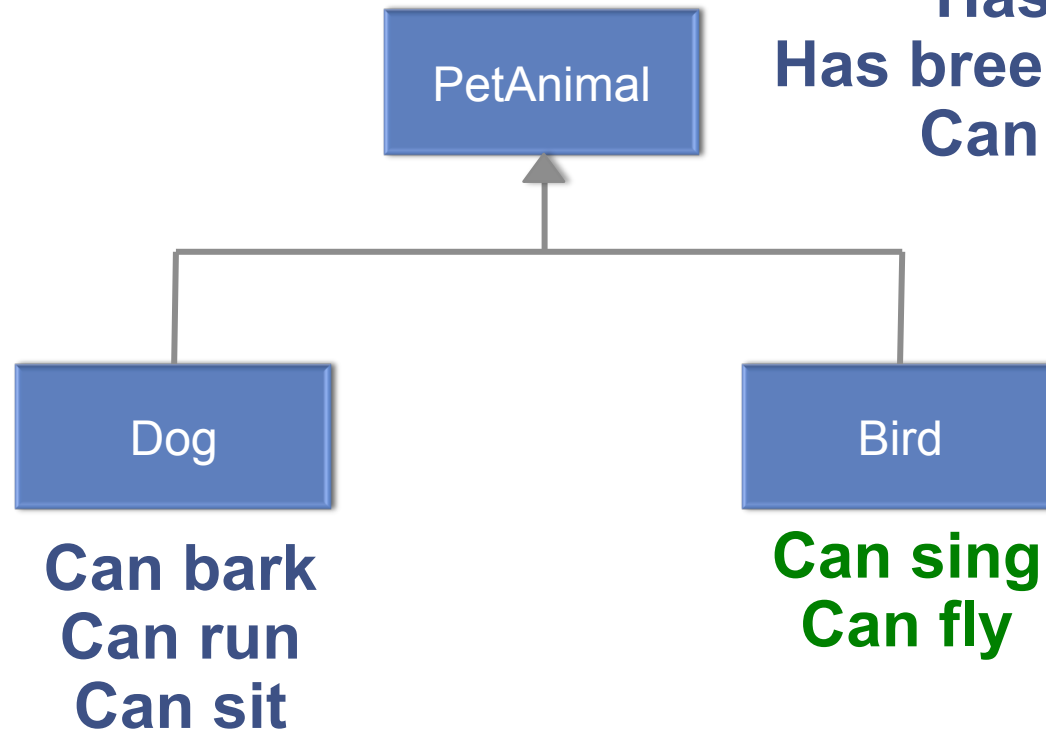
INHERITANCE

O-O INHERITANCE

Generic

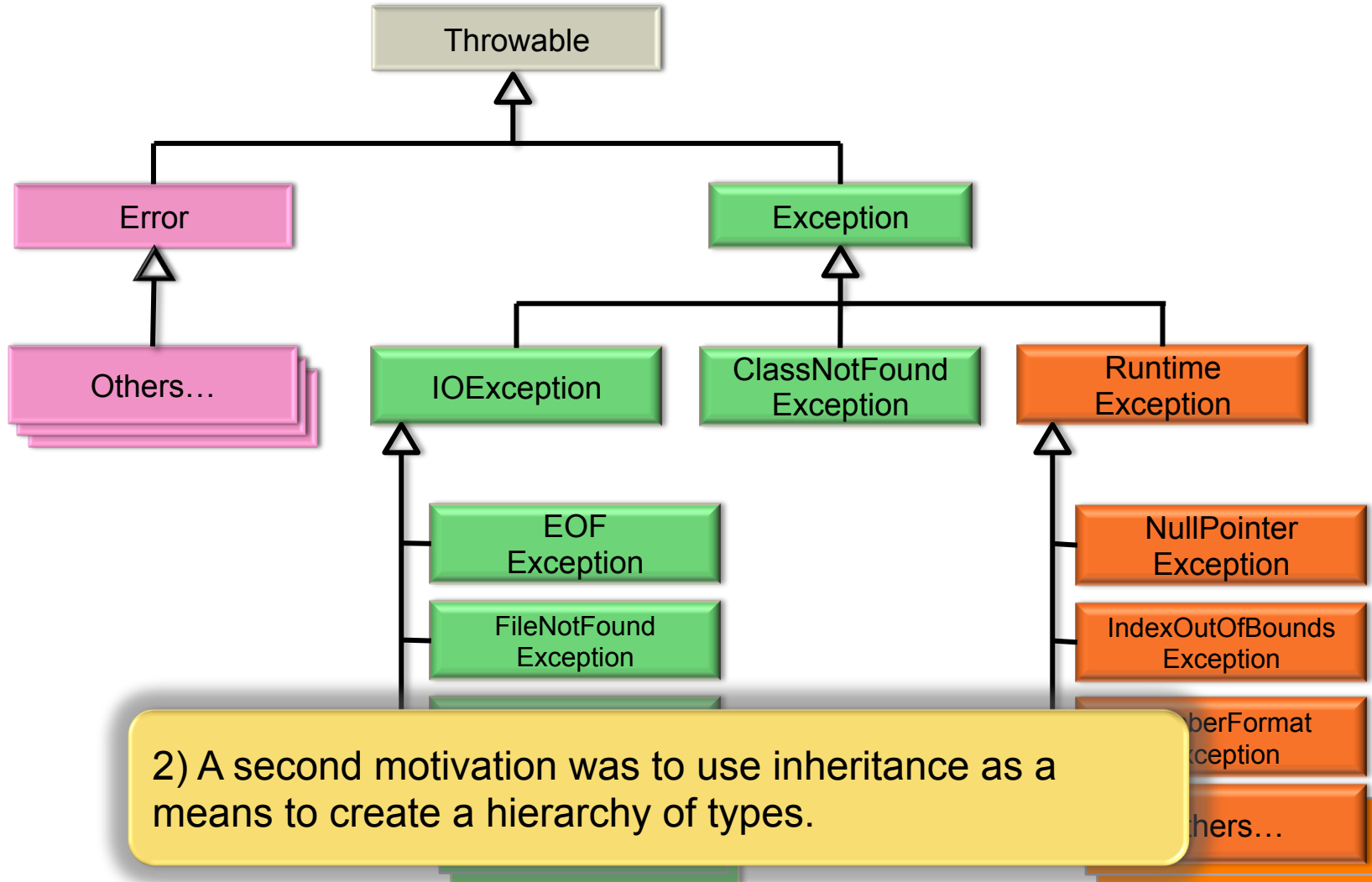


Specific



1) Initial motivation was to combine duplicate class methods into a common class.

EXCEPTION HIERARCHY

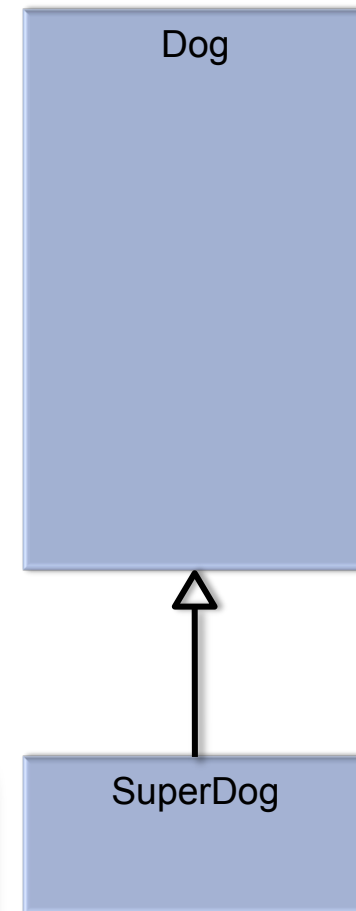


ALTERING A CLASS

- Sometimes an existing class is almost exactly what you want, if you could just change it a little
 - Altering existing functionality/methods
 - Adding new functionality/methods

override/change an existing method
use and add to an existing method
add new methods

3) A third motivation is to use inheritance to create new or improved versions of existing classes



OBJECT-ORIENTED

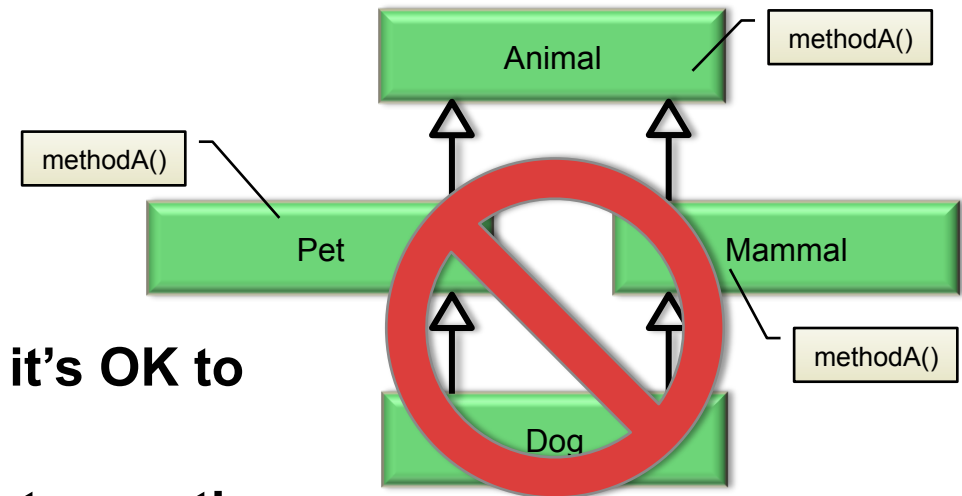
EXTENDS

OVERRIDING METHODS

EXTENDS DEMO

THE MULTIPLE INHERITANCE PROBLEM

extends ONLY one



- In some language (not Java), it's OK to inherit from multiple classes
- This mostly works, except that sometimes you can get into trouble
 - If a class inherits from two different classes, and they inherit from the same class, the compiler can get confused
 - This is known as '**Diamond Inheritance**'
 - To avoid this problem, Java bans multiple inheritance from a single class

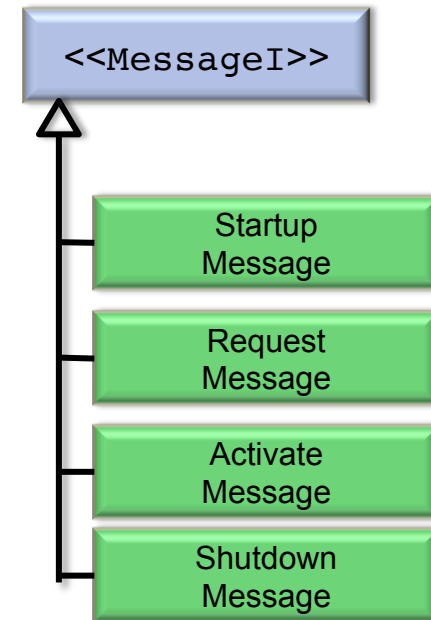
OBJECT-ORIENTED

INTERFACES

implements multiple

INTERFACES

- **A class may conform to an interface specification by implementing an interface**
 - Altering existing functionality/methods
 - Adding new functionality/methods
- **While methods are defined, there are no implementation of methods within the interface class**
- **An interface may define static final variables**
- **An interface may extend other interfaces**



INTERFACES

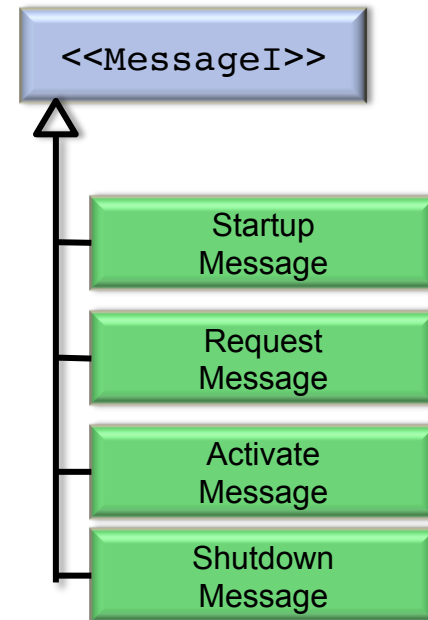
An interface class example:

```
package com.xyzcorp.io;

public interface MessageI {
    public static final SEND = 1;
    public static final RECV = 2;
    public int getSize();
}
```

```
package com.xyzcorp.io;

public class RequestMessage implements MessageI {
    public int senderId;
    private String message;
    ...
    public int getSize() {
        return(4 + message.length() * 2); // Integer: 4 bytes, Unicode: 2
    }
    ...
}
```



OBJECT-ORIENTED

INTERFACE DEMO

OBJECT-ORIENTED

ABSTRACT CLASSES

ABSTRACT

- Sometimes you want to let people write their own classes, but you'd like to give them a good starting point
- Abstract classes lie between a fully complete class, and an interface that only defines methods
- **Abstract classes**
 - May have complete methods
 - May define variables
 - May create method signatures that must be implemented by others
 - Are not complete, so they cannot be instantiated until they are completed by an inheriting class

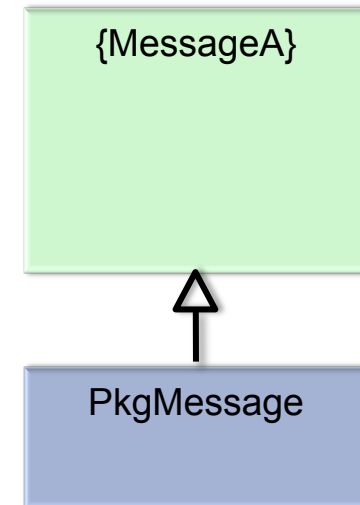
ABSTRACT

An **abstract** class example:

```
package com.xyzcorp.io;

public abstract class MessageA {
    public int senderId;
    private String message;

    public int getSenderId() {
        if (senderId > 1024) return senderId;
        else return 0;
    }
    ...
    public abstract void send();
    public abstract byte[] marshal();
    public abstract void parse(ByteBuffer msgBuf);
    ...
}
```



OBJECT-ORIENTED

ABSTRACT CLASS EXAMPLE

INTRO TO SINGLETON

DESIGN PATTERNS

SINGLETON

Whenever you have a class that will only exist once in a system, you can use a Design Pattern called **Singleton** to easily create it.

```
public class MyCentralData {  
  
    private static MyCentralData instance = null; // only one  
  
    private MyCentralData() { } // A private constructor  
  
    public static MyCentralData instance() {  
        // Creation only happens the first time  
        if (instance == null) instance = new MyCentralData();  
        return (instance); // All other times we get the first one  
    }  
  
    ...  
    public int getData() { return 0; } // Misc. routines  
}
```

FILE: READING THE FILESYSTEM

FILE I/O INTRO

FILE

- **In Java, a File instance is treated as a pointer to a disk directory index**
 - It might be a file or a folder
 - It might exist, or might need to be created
 - It might be a single file, or a path to a file/folder
- **A File isn't about reading or writing, it's about locating a resource on the disk**
 - Once you have that you can get a Reader or a Writer
- **Convenience class like FileWriter and FileReader do the job of many classes**
 - File
 - BufferedReader / BufferedWriter
 - InputStream / OutputStream

FILE

Using File to locate an existing disk file

Imports File class

```
import java.io.*;

public File getDataFile(String filename) throws IOException {
    File file = new File(filename);
    if (!file.exists())
        throw new IOException("No file found");
    if (file.isDirectory())
        throw new IOException("File is directory");
    if (!file.canRead())
        throw new IOException("File not readable");
    return file;
}
```

FILE I/O DEMO

LOGGING

LOGGING

- **Logging produces standardized messages that may be printed to the screen or archived into files**
`2015-10-14T13:14:49EDT - INFO: Registered Entity builder`
- **Log Messages are rated by severity (they have Levels)**
- **Log messages may be routed using Handler classes, which act as a queue**
 - ConsoleHandler
 - FileHandler
 - StreamHandler, etc.
- **A configuration file may be used to specify the handling and display of Log messages**
- **Log4j is a popular open source logger from Apache Foundation**
- **In our demo, we'll be using the built in Java logger package**

LOGGING

For any class, you may add a logger:

```
import java.util.logging.*;

public class MyClass {

    private static Logger log =
        Logger.getLogger(MyClass.class.getName());

    public MyClass() {
        log.info("Constructing a MyClass instance");
    }
    ...
    public int calcData(int val) {
        if (val < 0)
            log.warning("Negative values encountered");
        return val * val;
    }
}
```

A logger may be named using a literal string, but it's common to use the fully qualified class name

LOGGING

- Logging messages are assigned levels
 - Finest, Finer, Fine, Config, Info, Warning, Severe

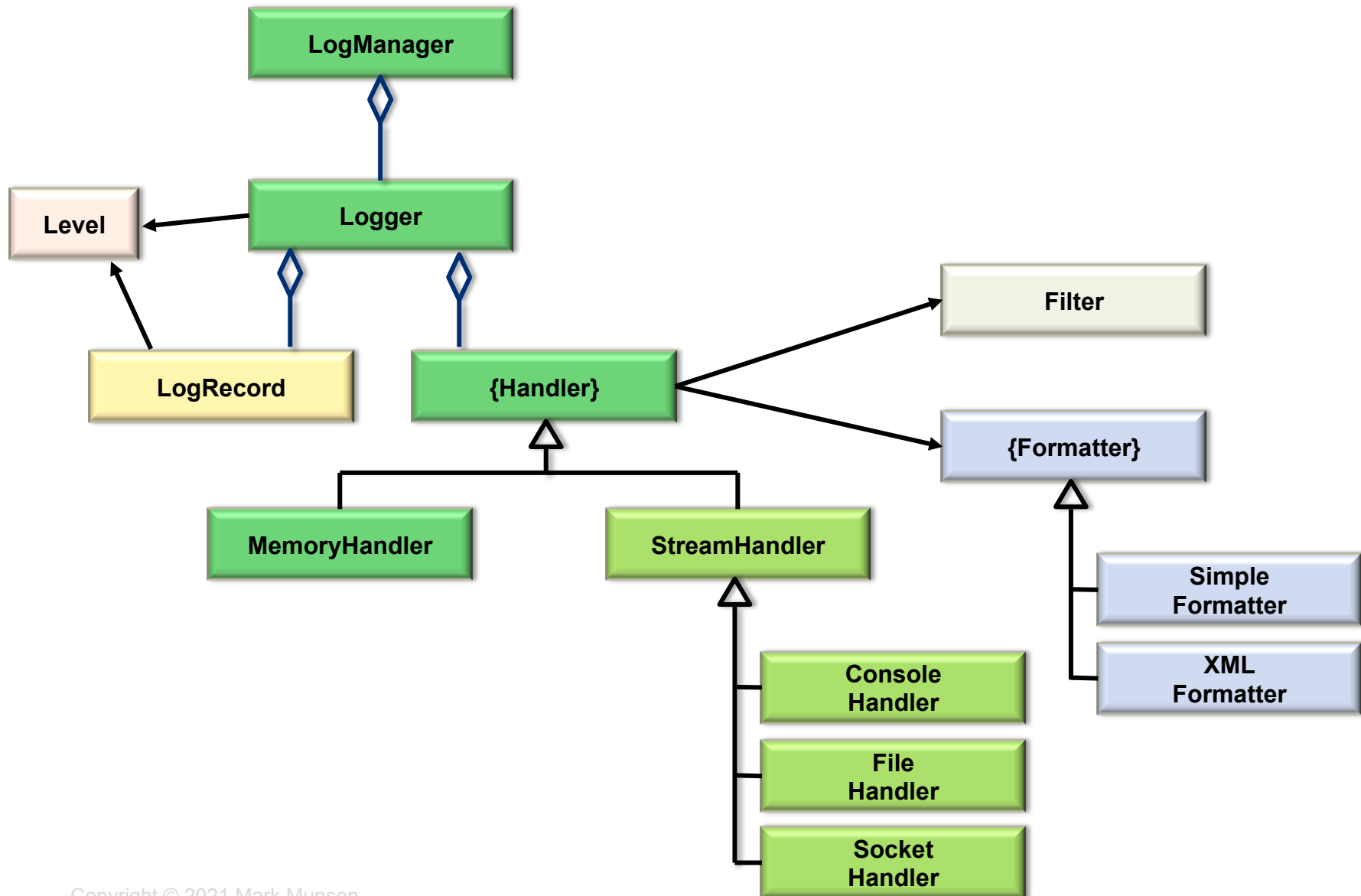
- Examples:

```
log.info("Regular information"); // level 800
log.warning("A cause for concern"); // level 900
log.severe("A big problem"); // level 1000
```

- Loggers can be assigned custom handlers

```
Handler handler = new FileHandler("server.log");
Logger.getLogger("").addHandler(handler);
```

LOGGING CLASS DIAGRAM



FILE LOGGER EXAMPLE

```
String sep = File.separator;
String logPath = ".." + sep + "logs" + sep + "server.log";

// Let's send all of the logging to a rotating disk file that uses stock XML formatting
try {

    File logDirFile = new File(logDirPath);
    if (!logDirFile.exists()) // If this log folder doesn't exist, create it
        logDirFile.mkdirs();

    // Create a rotating logfile handler and add it to our logger
    Handler handler2 = new FileHandler(logPath, LOG_SIZE, LOG_ROTATION_COUNT);
    Logger.getLogger("").addHandler(handler2);

} catch (SecurityException e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

LOGGING CONFIG FILE

log.properties:

```
handlers=java.util.logging.ConsoleHandler
.level=INFO
java.util.logging.ConsoleHandler.level=INFO
java.util.logging.ConsoleHandler.formatter=java.util.
logging.SimpleFormatter
edu.neu.csye6200.myproject.level=FINE
```

To configure on startup, specify a logging properties file to use

```
> java ... -Djava.util.logging.config.file=../conf/log.properties ...
```

LOGGING DEMO

PRE-MIDTERM

QUIZ 2 ONLINE