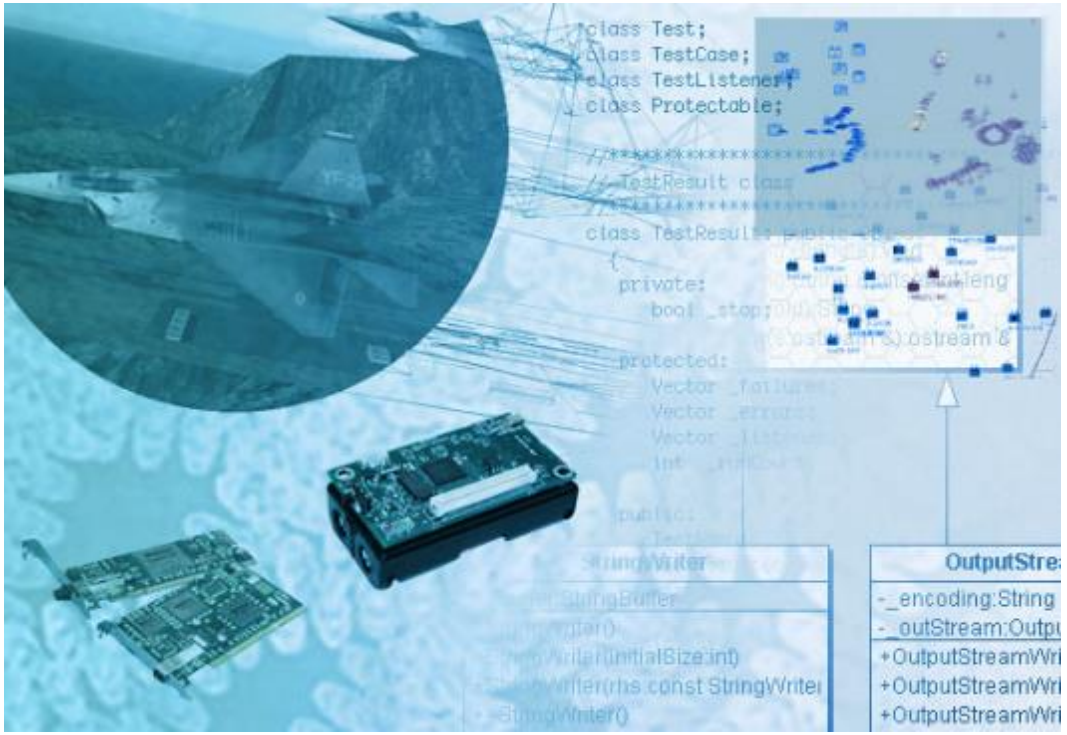
CSYE 6200

CONCEPTS OF OBJECT-ORIENTED DESIGN

FALL 2016 – WEEK 6

MARK G. MUNSON



ADMINISTRATION

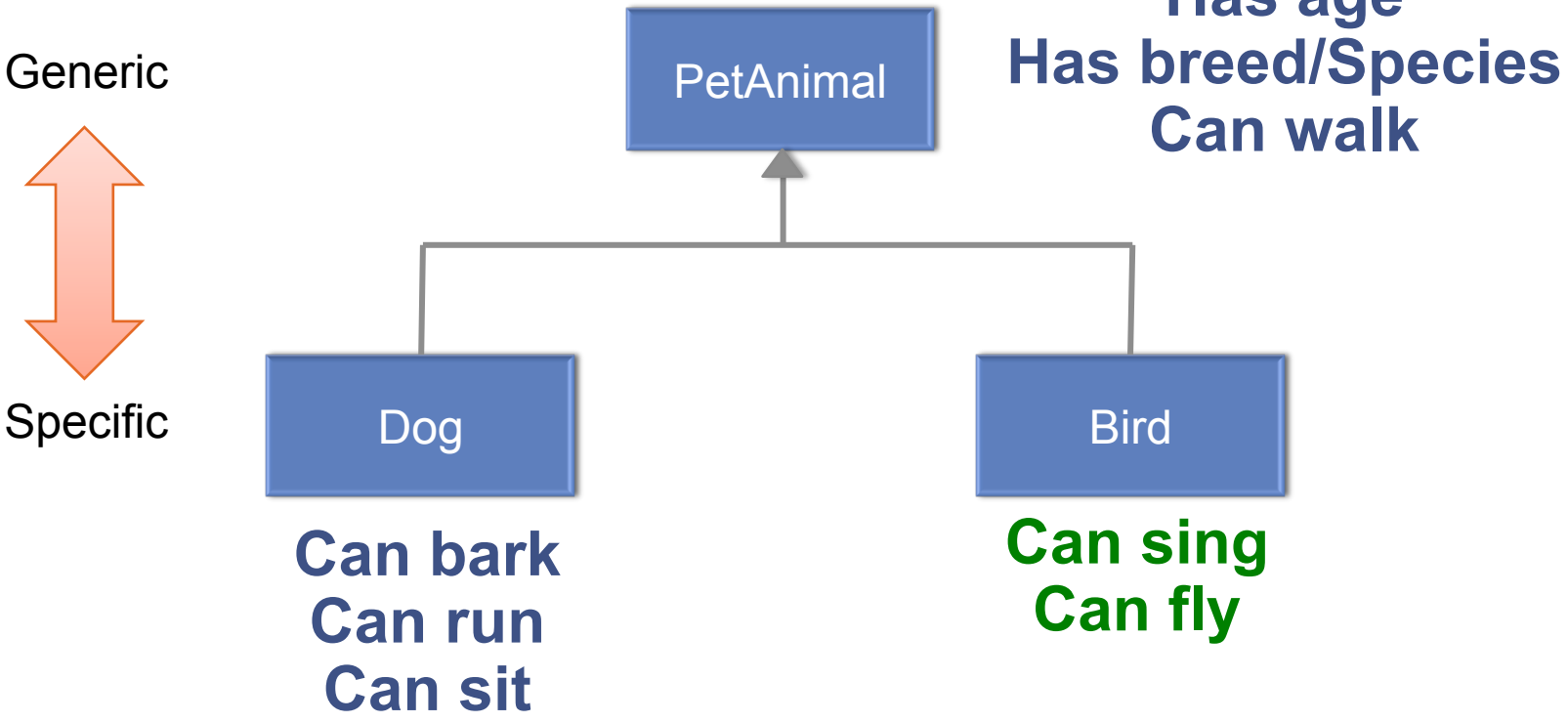
- **Auditing is not permitted** – *Only registered students may attend lectures*
- **Assignment #3 due October 10/11** – *Today!*
  - Code and sample output in Blackboard
- **Quiz next week – October 18/19**
  - Similar to last quiz
  - Covers Ch. 1 – 10
- **Midterm Exam on week 8 (October 25/26)**
  - Extra review next week – will cover:
    - Ch. 1 – 10
    - UML Class Diagrams
    - Coding

THE LECTURE

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

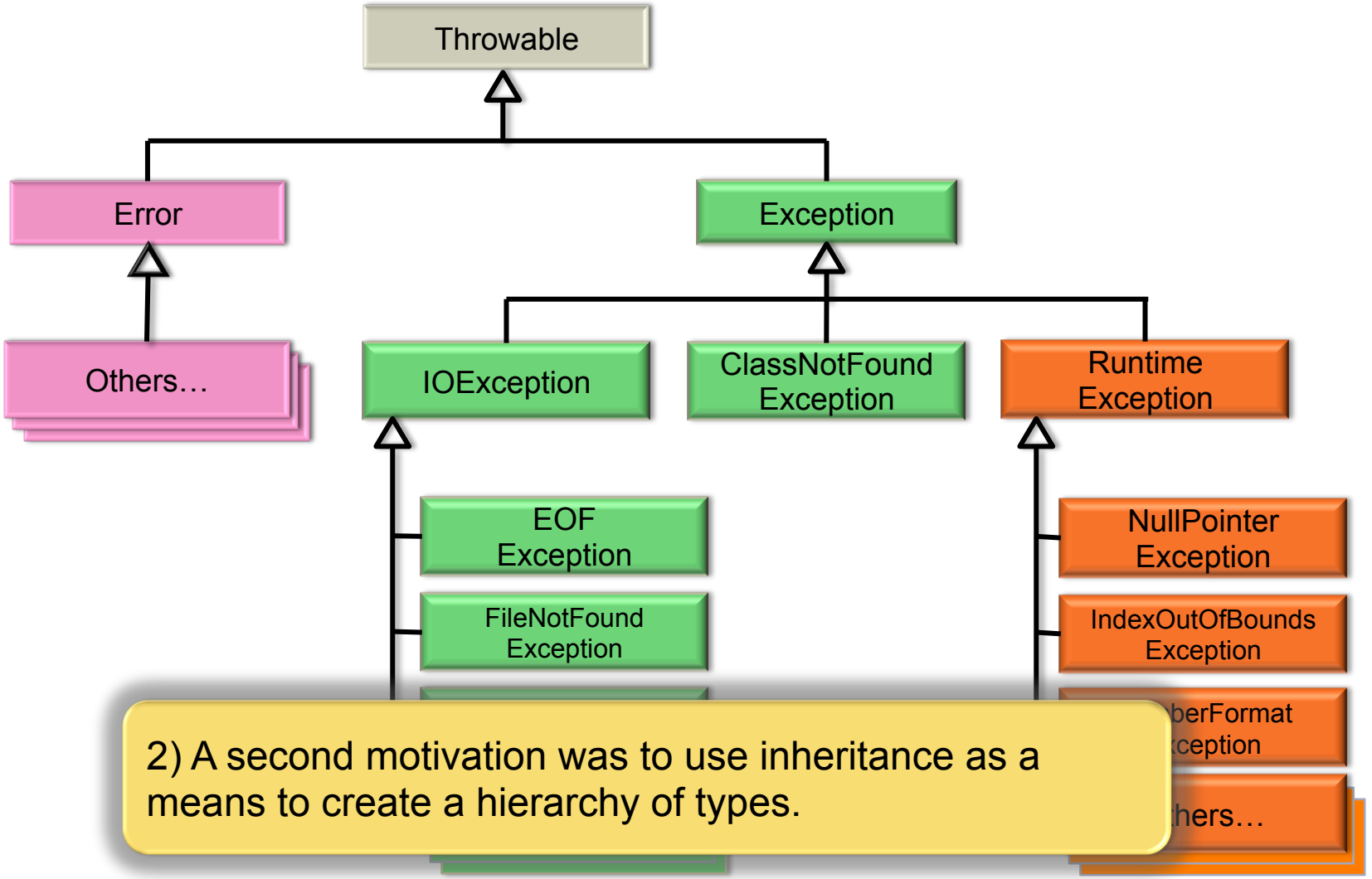
INHERITANCE

OO INHERITANCE



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

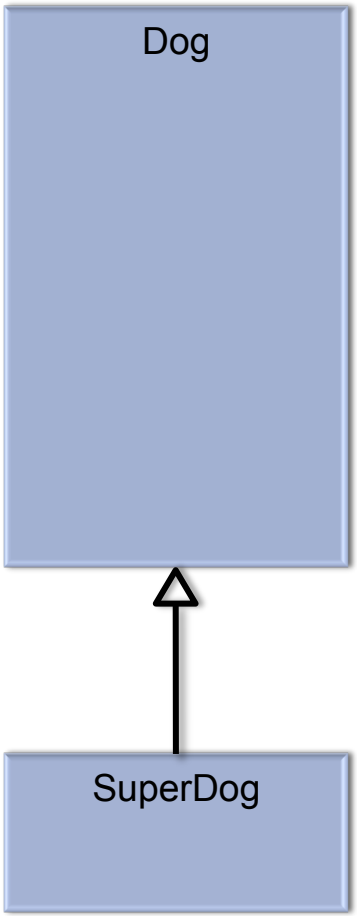
EXCEPTION HIERARCHY



2) A second motivation was to use inheritance as a means to create a hierarchy of types.

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



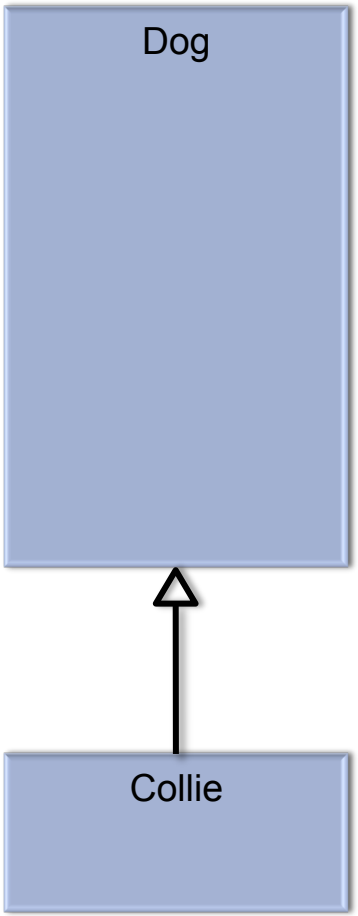
3) A third motivation is to use inheritance to create new or improved versions of existing classes

OBJECT ORIENTED

EXTENDS

EXTENDS

- **A class may be extended solely to create a new class, which changes an existing class by**
  - Altering existing functionality/methods
    - Overriding existing methods
  - Adding new functionality/methods
    - Adding new methods
    - Overriding methods while calling existing implementations (super.method())



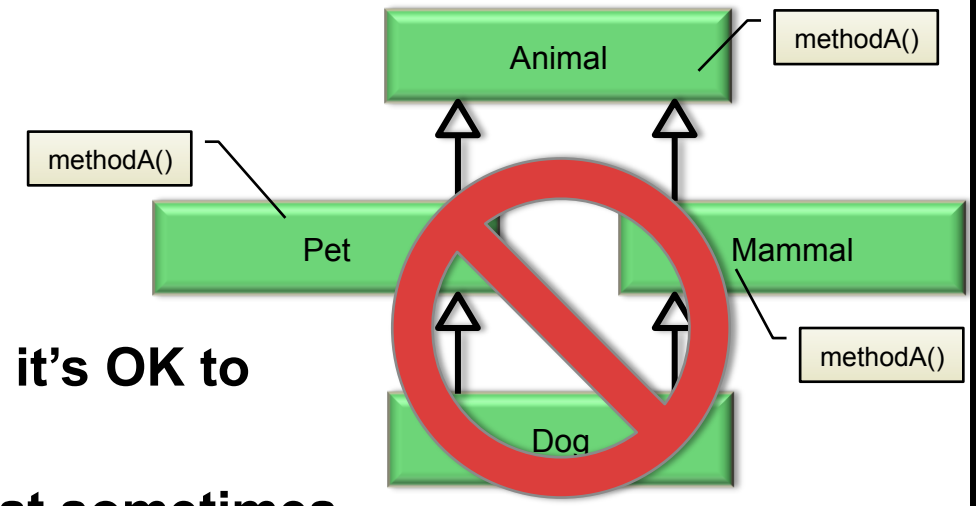
THE 'IS-A' AND 'HAS-A' TEST

- **If a class you are designing actually 'is-a' more specific version of another existing class, then you can use inheritance to `extend` from the other class.**
  - Be careful – a changing parent class can break a child
  - Use it judiciously – it tends to violate encapsulation

- **If instead, your class 'has-a' instance of another class, then you should just own member variables of that other class.**
  - Composition allows you to control method exposure

- **Before extending another class, as yourself this question:**
  Do you want to extend any flaws from that class?

THE MULTIPLE INHERITANCE PROBLEM



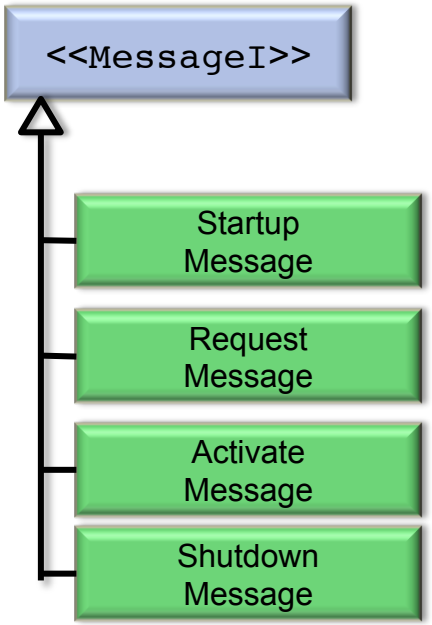
- **In some language (not Java), it's OK to inherit from multiple classes**
- **This mostly works, except that sometimes you can get into trouble**
  - If a class inherits from two different classes, and they inherit from the same class, the compiler can get confused
  - This is known as 'Diamond Inheritance'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

OBJECT ORIENTED

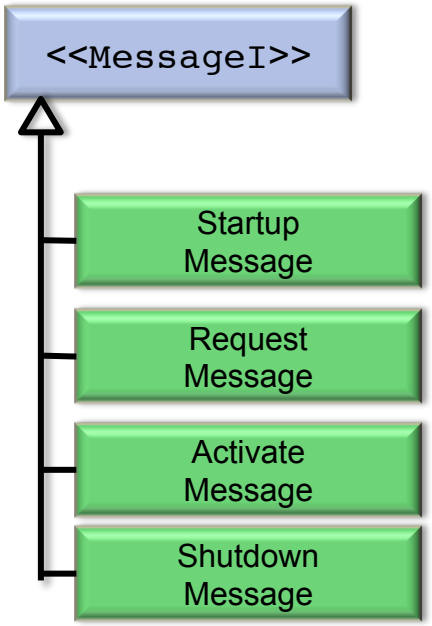
INTERFACES

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
   }
…
}
```

INTERFACES

- Allow for multiple inheritance while side-stepping Java's limit on single inheritance extension
- Are light-weight
- Are easy to code, and don't convey coding errors
- Provides a contract for Application Programmer Interface (API) development, while permitting full flexibility with code implementation

OBJECT ORIENTED

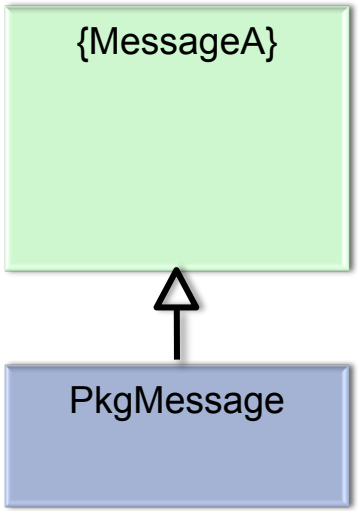
ABSTRACT CLASSES

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);
…
}
```

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 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;
    }
}
```

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 my be routed using Handler classes, which act as a queue**
  - ConsoleHandler
  - FileHandler
  - StreamHandler, etc.
- **A configuration file may be used to specify the handing 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

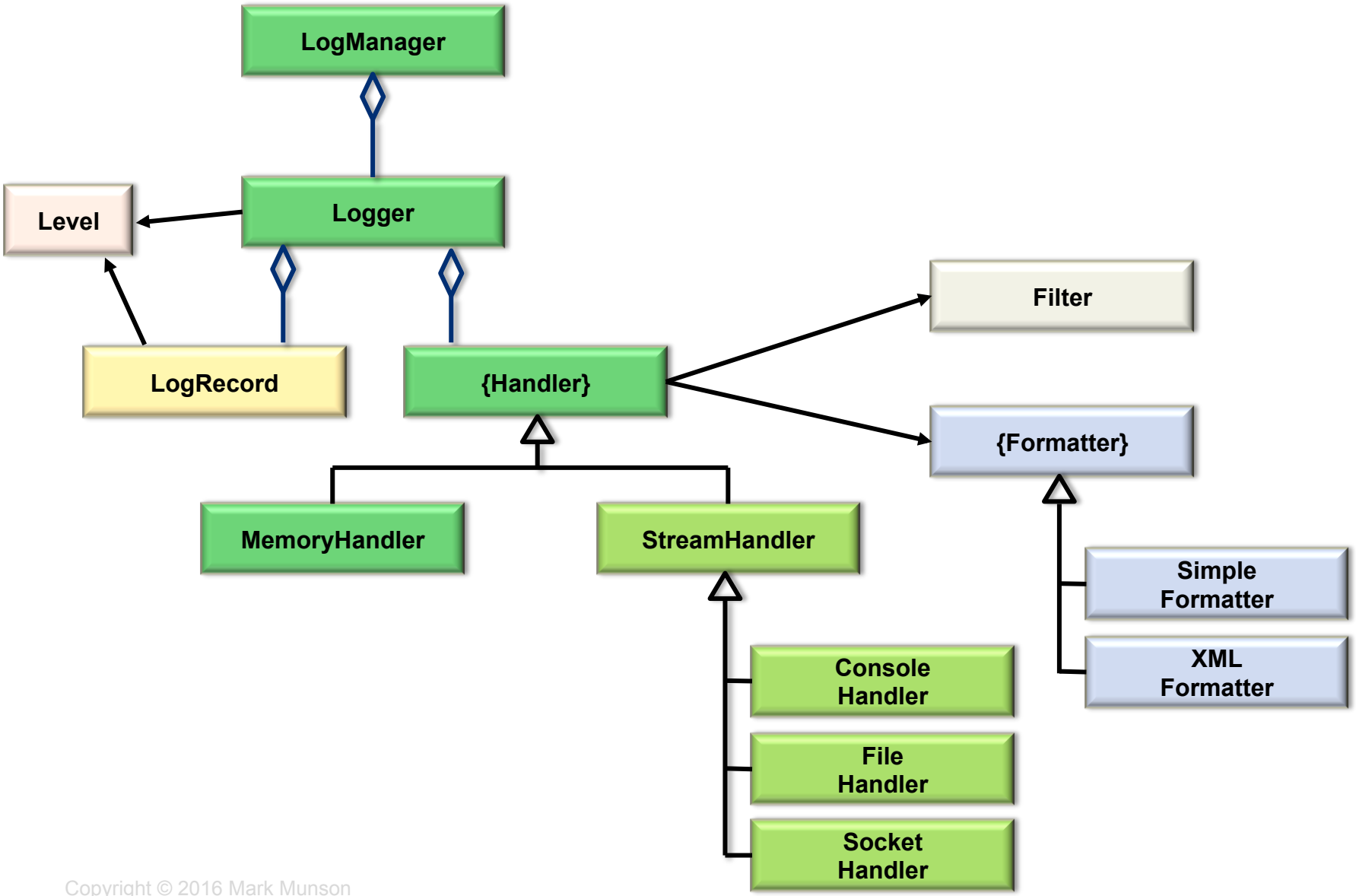
- **Logging messages are assigned levels**
  - Finest, Finer, Fine, Config, Info, Warning, Severe
- **Examples:**

```
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



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.csye6200.myproject.level=FINE
```

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

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

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();
}
```

NEXT WEEK / EXAM PREP

JABG: Read

- Ch. 13 Generics

- **Assignment #4 Due next week – October 18/19**
  - Create a package called edu.neu.csye6200.registry, and move all related assignment 2/3 code into it
  - Convert your VehicleRegistry class to use the Singleton pattern
    - Add a method to VehicleRegistry that sorts your vehicles by license
      - Ensure that you have at least 10 Vehicles defined
      - Print the results
  - Add logging to your VehicleRegistry an RegistryIO classes
    - Log information messages for class creation, and for load/save methods
    - Log a severe message if an error is captured in your try-catch block(s)
    - Add a FileHandler to send your log messages to disk

- **Quiz next week**
  - Can you answer self test questions for Ch. 1 – 10?
    - Answers are in the back – Appendix A
  - Can you draw UML class diagrams?
    - Class detail
    - Association connectors (inheritance, collections)