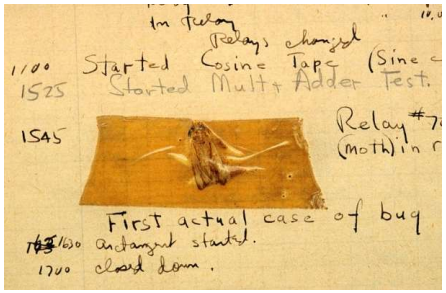
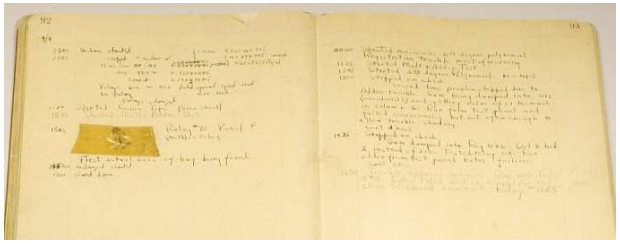
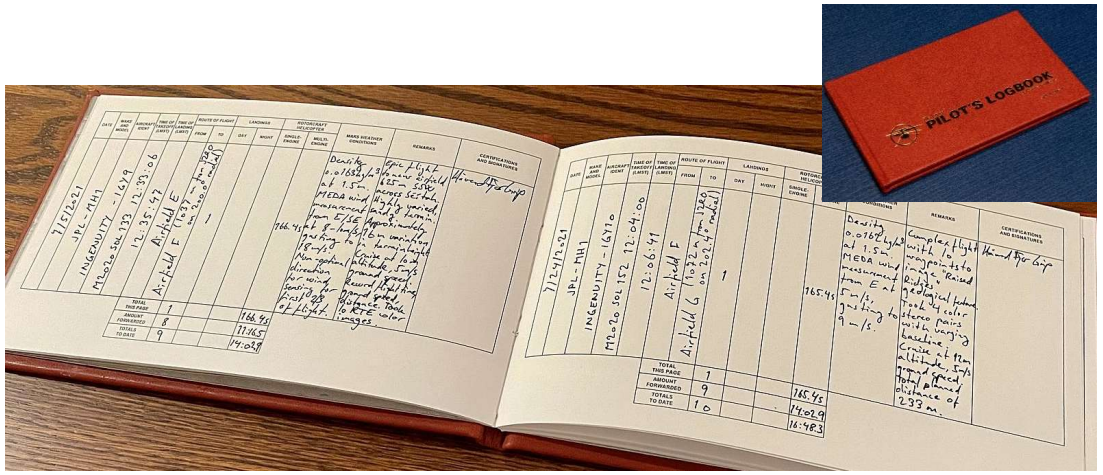
java.util.logging
(un framework de logging en Java)





1947, Loogbook, Computador Mark II



Logbook usado para el helicóptero Mars Ingenuity de la NASA

Logging (propósito)

- Agregamos código de loggin a nuestra aplicación para entender lo que pasa con ella, por ejemplo:
  - Reportes de eventos importantes, errores y excepciones
  - Pasos críticos en la ejecución
  - Inicio y fin de operaciones complejas o largas
- Los logs son útiles para desarrolladores, administradores y usuarios
- Comentario al margen: ¡Los logs no reemplazan al testing!

Herramientas y estrategias

- Podemos utilizar System.out.println, pero ...
- Los frameworks/APIs de logging permiten ...
  - Estandarizar la práctica
  - Activar o desactivar logs selectivamente, incluso sin modificar el código
  - Enviar los reportes a distintos formatos y destinos (txt, JSON, XML, archivos, pantalla, sockets ...)
  - Oculta los detalles de implementación

Java Logging Framework

- Framework de logging, incluido en el SDK define:
  - El log se hace enviando mensajes a objetos Logger
  - Como se crean, organizan y recuperan esos objetos
  - Como se configuran
  - Como se activan y desactivan
  - A que prestan atención y a que no
  - Cómo se formatean los logs
  - A donde se envían los logs

Java Logging Framework

- El framework se encarga del registro de Loggers
- Los loggers se organizan en un espacio jerárquico de nombres
  - Heredan propiedades, y propagan mensajes
- Los mensajes de error se asocian a niveles (importancia)
  - Permite filtrar mensajes por nivel
- Permite enviar mensajes a consola, archivos, sockets
- Permite ajustar el formateo de los mensajes (XML, texto)
- Se puede extender (nuevos formatos, destinos, filtros ...)

Ejemplo básico (1)

```java
public class Sandbox {
    public static void main(String[] args) throws IOException {
        Logger.getLogger("app.main").addHandler(new FileHandler("log.txt"));
        Logger.getLogger("app.main").log(Level.INFO, "App iniciada");
        try {
            // Acá que hace algo que "podría" resultar en una excepción
            int explodesForSure = 1 / 0;
        } catch (Exception ex) {
            Logger.getLogger("app.main").log(Level.SEVERE, "Explotó!", ex);
        }
        Logger.getLogger("app.main").log(Level.INFO, "App terminada");
    }
}
```

Ejemplo básico (1)

```
public class Sandbox {
    public static void main(String[] args) throws IOException {
        Logger.getLogger("app.main").addHandler(new FileHandler("log.txt"));
    }
}
```

La configuración de los loggers suele ocurrir al iniciar la aplicación (y se mantiene globalmente).

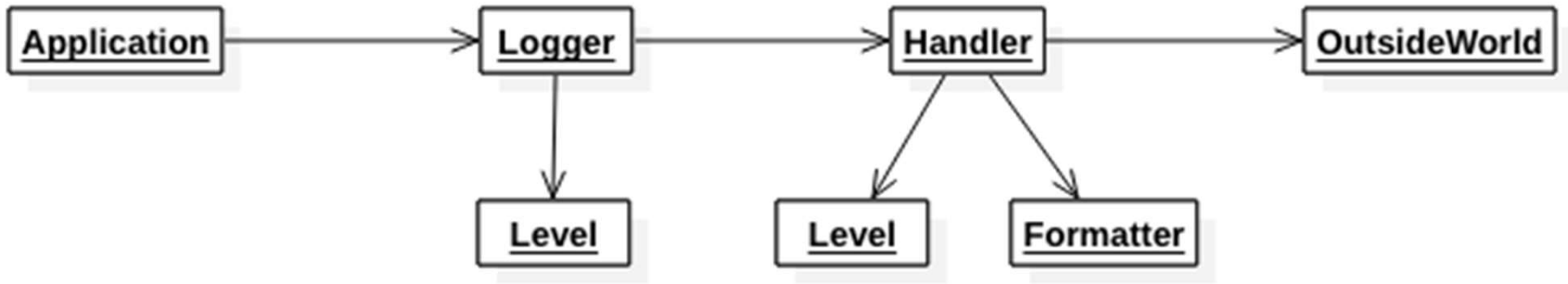
Ejemplo básico (1)

En cualquier método, se puede recuperar y utilizar un logger

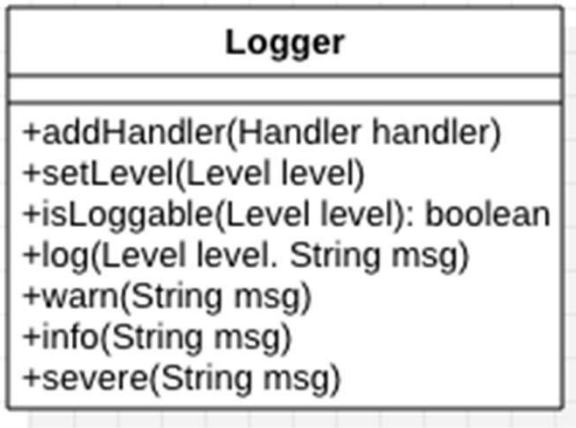
```
    } catch (Exception ex) {
        Logger.getLogger("app.main").log(Level.SEVERE, "Explotó!", ex);
    }
    Logger.getLogger("app.main").log(Level.INFO, "App terminada");
}
}
```

Partes básicas (visibles)

- Logger: objeto al que le pedímos que emita un mensaje de log
- Handler: encargado de enviar el mensaje a donde corresponda
- Level: indica la importancia de un mensaje y es lo que mira un Logger y un Handler para ver si le interesa
- Formater: determina como se ”presentará” el mensaje



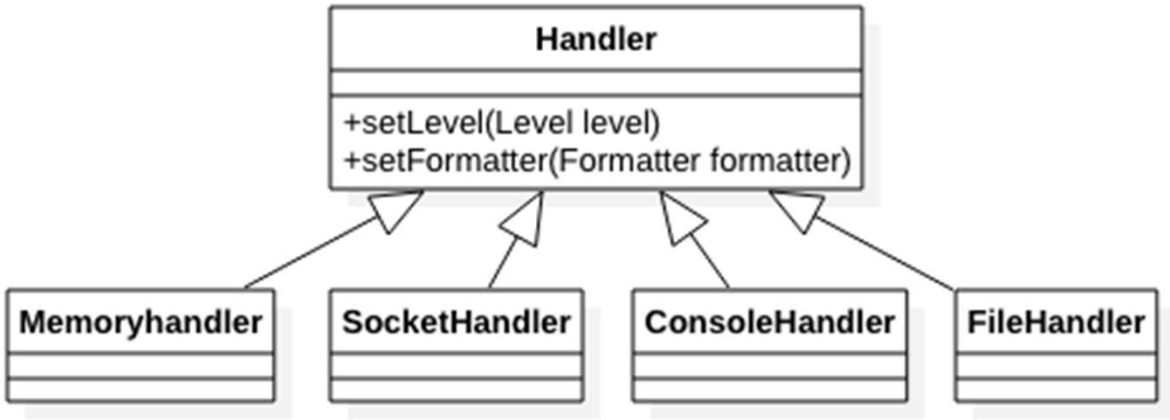
Logger



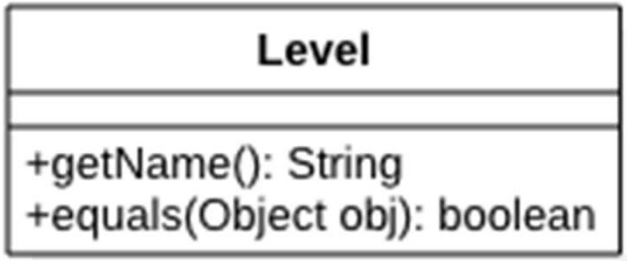
- Podemos definir tantos como necesitemos
  - Instancias de la clase Logger
  - Las obtengo con Logger.getLogger(String nombre)
- Cada uno con su filtro y handler/s
- Se organizan en un árbol (en base a sus nombres)
  - Heredan configuración de su padre (handlers y filters)
- Envío el mensaje log(Level, String) para emita algún mensaje
  - Alternativamente uso warn(), info(), severe() …

Handler

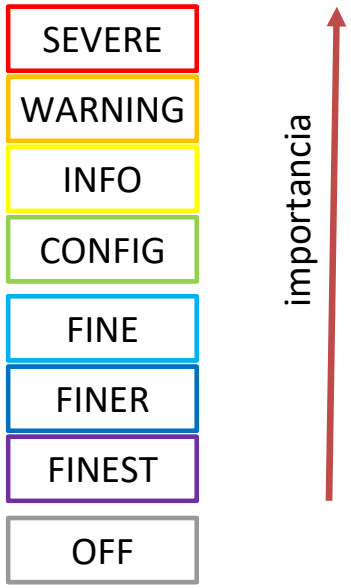
- Recibe los mensajes del Logger y determina como “exportarlos”
- Instancias de MemoryHandler, ConsoleHandler, FileHandler, o SocketHandler
- Puede filtrar por nivel
- Tiene un Formatter



Level

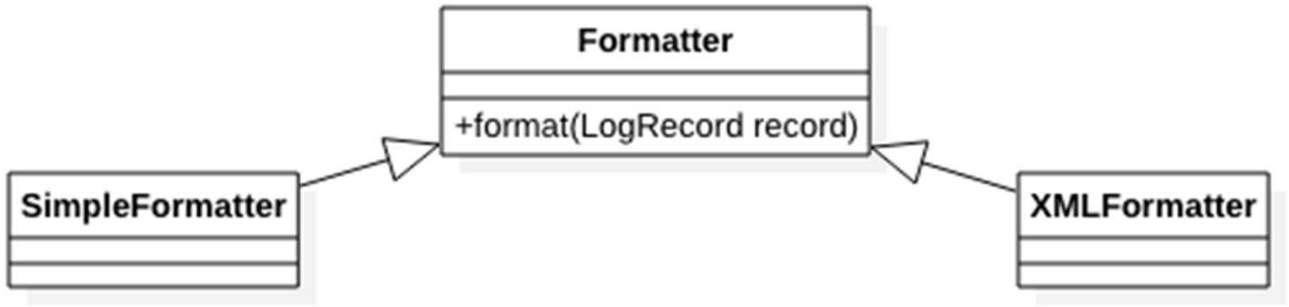


- Representa la importancia de un mensaje
- Cada vez que pido que se loggee algo, debo indicar un nivel
- Los Loggers y Handler comparan el nivel de un cada mensaje con el suyo para decidir si les interesa o no
- Si te interesa un nivel, también te interesan los que son más importantes que ese
- Hay niveles predefinidos, en variables estáticas de la clase Level (p.e., Level.OFF)



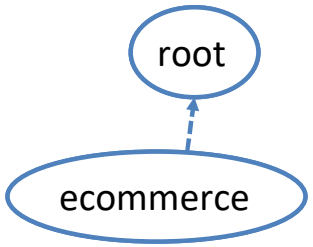
Formatter

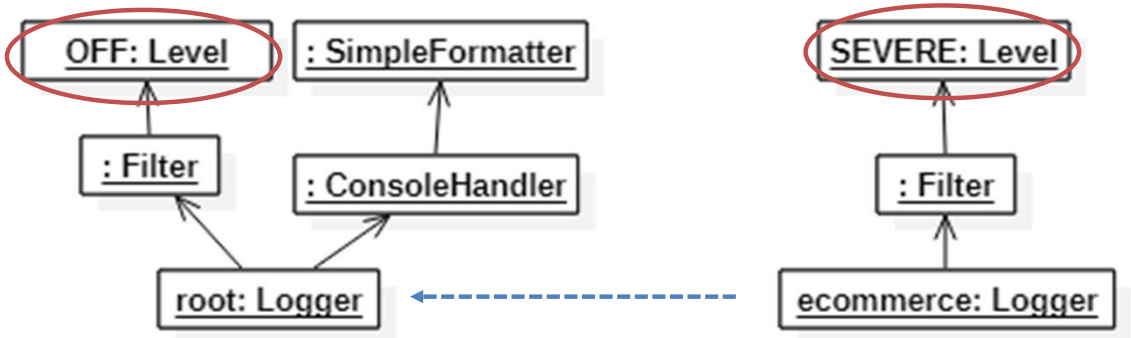
- El Formatter recibe un mesaje de log (un objeto) y lo transforma a texto
- Son instancias de: SimpleFormatter o XMLFormatter
- Cada handler tiene su formatter
  - Los FileHandler tienen un XMLFormatter por defecto
  - Los ConsoleHandler tienen un SimpleFormatter por defecto



Ejemplo avanzado (como caja negra)

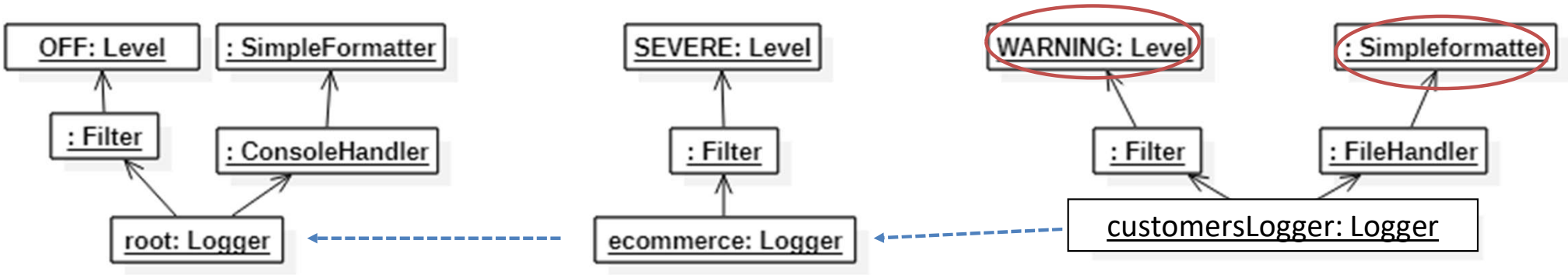
```
//Loggers apagados por defecto
Logger.getLogger("").setLevel(Level.OFF);

//Loggers encendidos en nivel SEVERE para ecommerce
//Utilizará un ConsoleHandler y un SimpleFormatter
Logger  ecommerce = Logger.getLogger("ecommerce");
ecommerce .setLevel(Level.SEVERE);
```





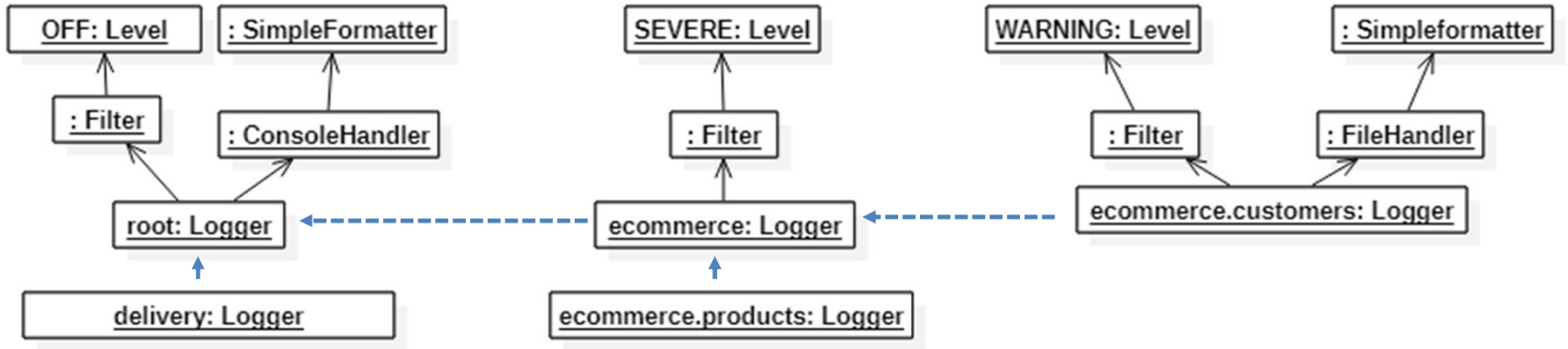
Ejemplo avanzado (como caja negra)

```
//Loggers apagados por defecto
```



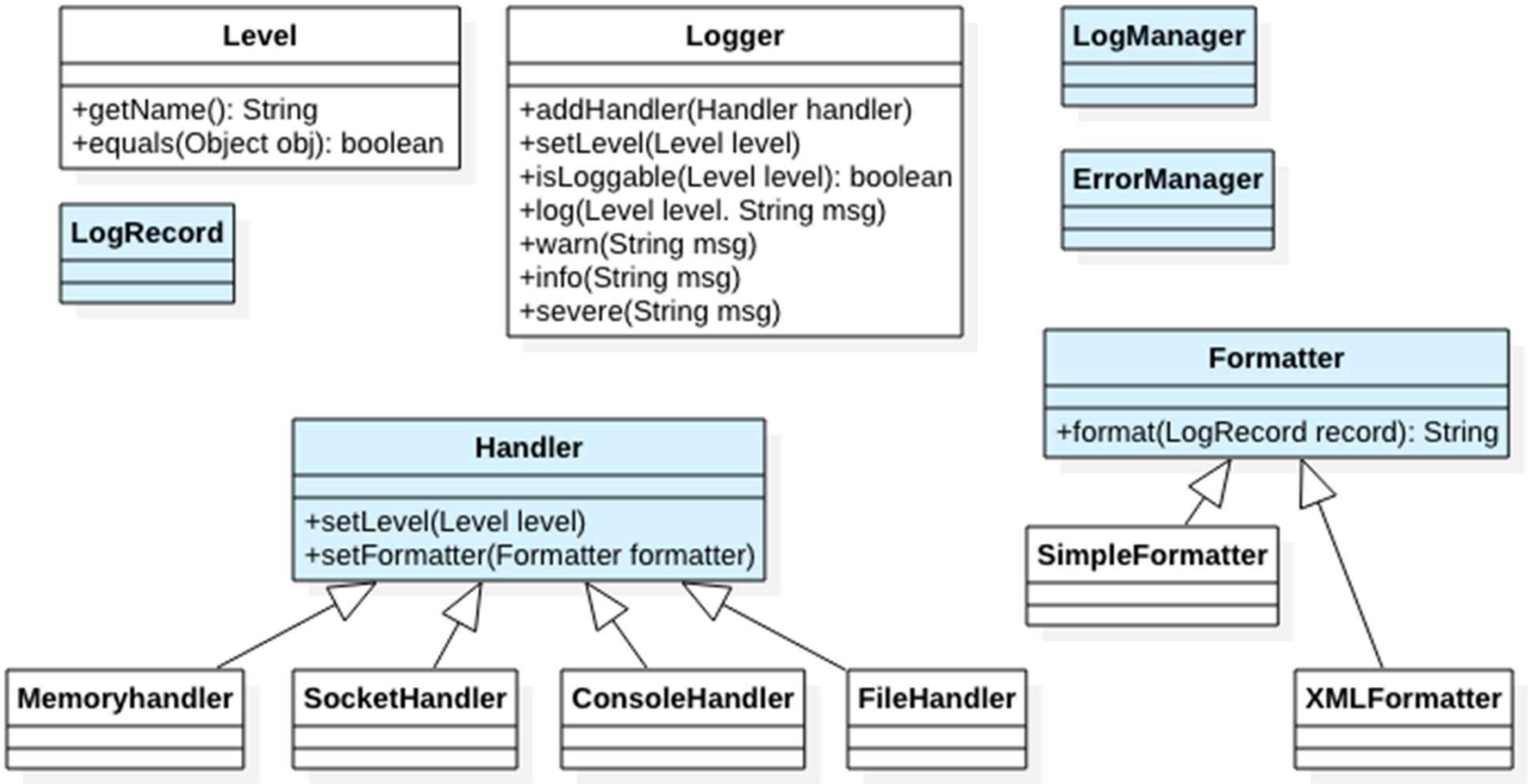

```
//Logger del servicio de customers de ecommerce, encendido en nivel WARNING
//con destino un archivo, en formato simple texto
Logger customersLogger = Logger.getLogger("ecommerce.customers");
customersLogger.setLevel(Level.WARNING);
FileHandler customersLoggerHandler = new FileHandler("ecommerce-customers.log");
customersLoggerHandler.setFormatter(new SimpleFormatter());
customersLogger.addHandler(customersLoggerHandler);
```

Ejemplo avanzado (como caja negra)




```
// Este logger hereda del raíz y no define nada propio, por lo tanto ignora el warning
Logger.getLogger("delivery").log(Level.WARNING, "Error in delivery");

// Este logger hereda de ecommerce por lo tanto ignora el warning
Logger.getLogger("ecommerce.products").log(Level.WARNING, "Stock inconsistency detected");

// A este logger le interesa el warning, que termina en un archivo con formato simple
Logger.getLogger("ecommerce.customers").log(Level.WARNING, "Stock inconsistency detected");
```

Lo que no se ve (desde afuera de la caja)



Lo que no se ve (desde afuera de la caja)

- LogRecord: objeto que representa un mensaje
- LogManager: es el objeto que mantiene el árbol de loggers; hay un solo LogManager (global)
- Filter: Logger y Handler tienen un Filter para determinar si algo les intersa o no (el Filter conoce al Level)
- ErrorManager: los Logger pueden tener un ErrorManager para lidiar con errores durante el logging
- Clases abstractas Handler y Formatter
- … y todas las relaciones entre ellos …

Lo que no cambia (frozenspot)

- Diseño / propuesta sobre como integrar logging
- Estrategia general
  - Busco un logger para configurarlo o pedirle que emita un mensaje
  - Tengo Logger, Handler, Formatter, Filter, Level
- Cómo se organizan los loggers (árbol gestionado globalmente)
- Las configuraciones se heredan
- Qué pasa cuando le digo log() a un logger (colaboraciones)

Lo que si cambia (hotspots)

- Cuántos y cuales loggers utilizo
- Qué logger “hereda” de cuál
- Qué le interesa a cada logger (su filter)
- Qué handlers se asocian a cada logger
- Qué formatter tiene cada handler y que le interesa (su filter)

Hasta ahí, todo caja negra …

Extendiendo el framework (caja blanca)

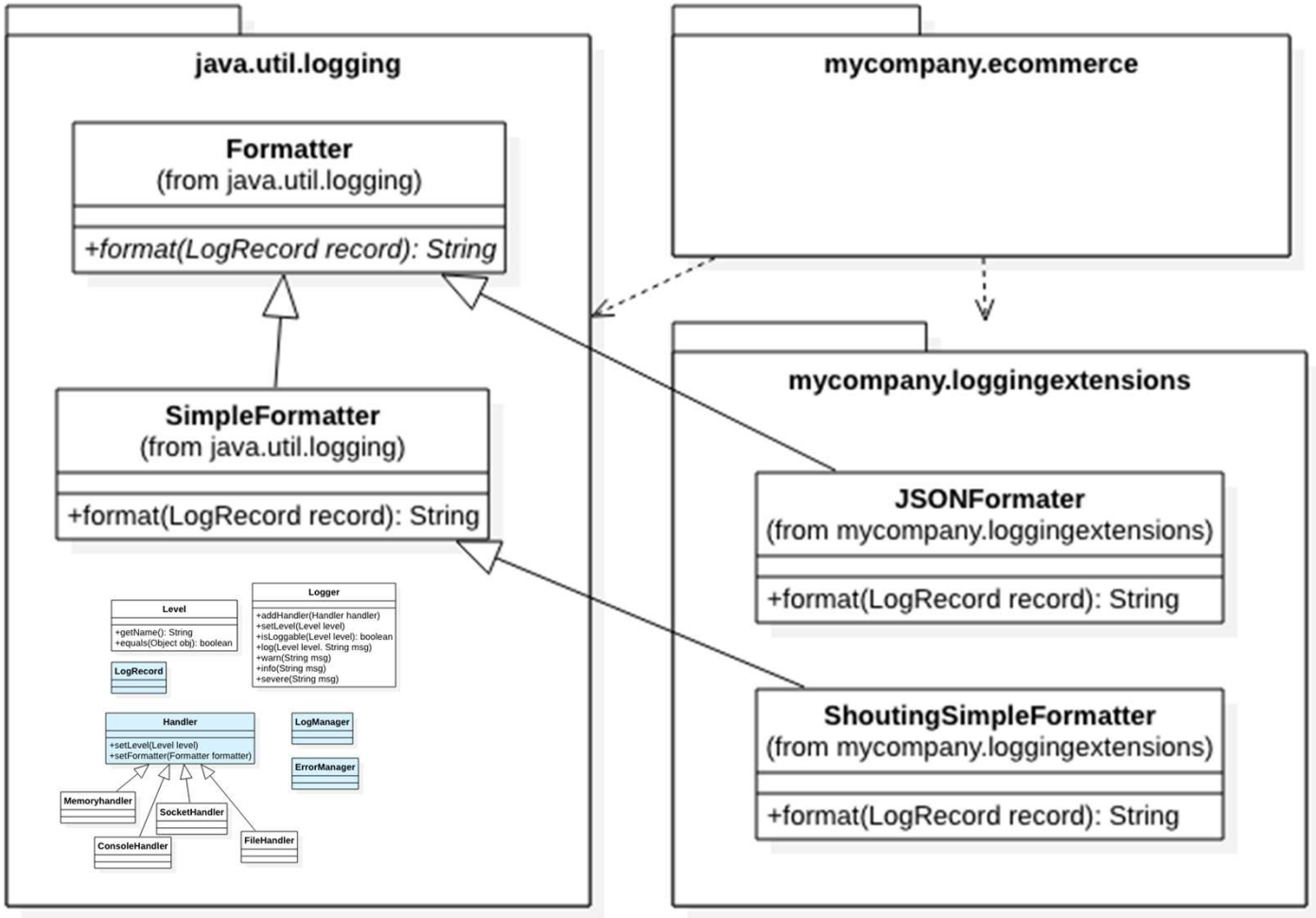
- Y, mirando adentro (caja blanca), puedo agregar nuevas clases de Formater, Handler y Filter
  - Nuevo Formatter: Subclasifico la clase abstracta Formatter o alguna de sus subclases
  - Nuevo Handler: Subclasifico la clase abstracta Handler o alguna de sus subclases
  - Nuevo Filter: Implemento la interfaz Filter
- Esto no es hacking, sino algo previsto por los diseñadores

Nuevos Formatters

```java
public class ShoutingSimpleFormatter extends SimpleFormatter {
    @Override
    public String format(LogRecord record) {
        // SHOUTING WITH ALL UPPERCASE
        return super.format(record).toUpperCase();
    }
}
```

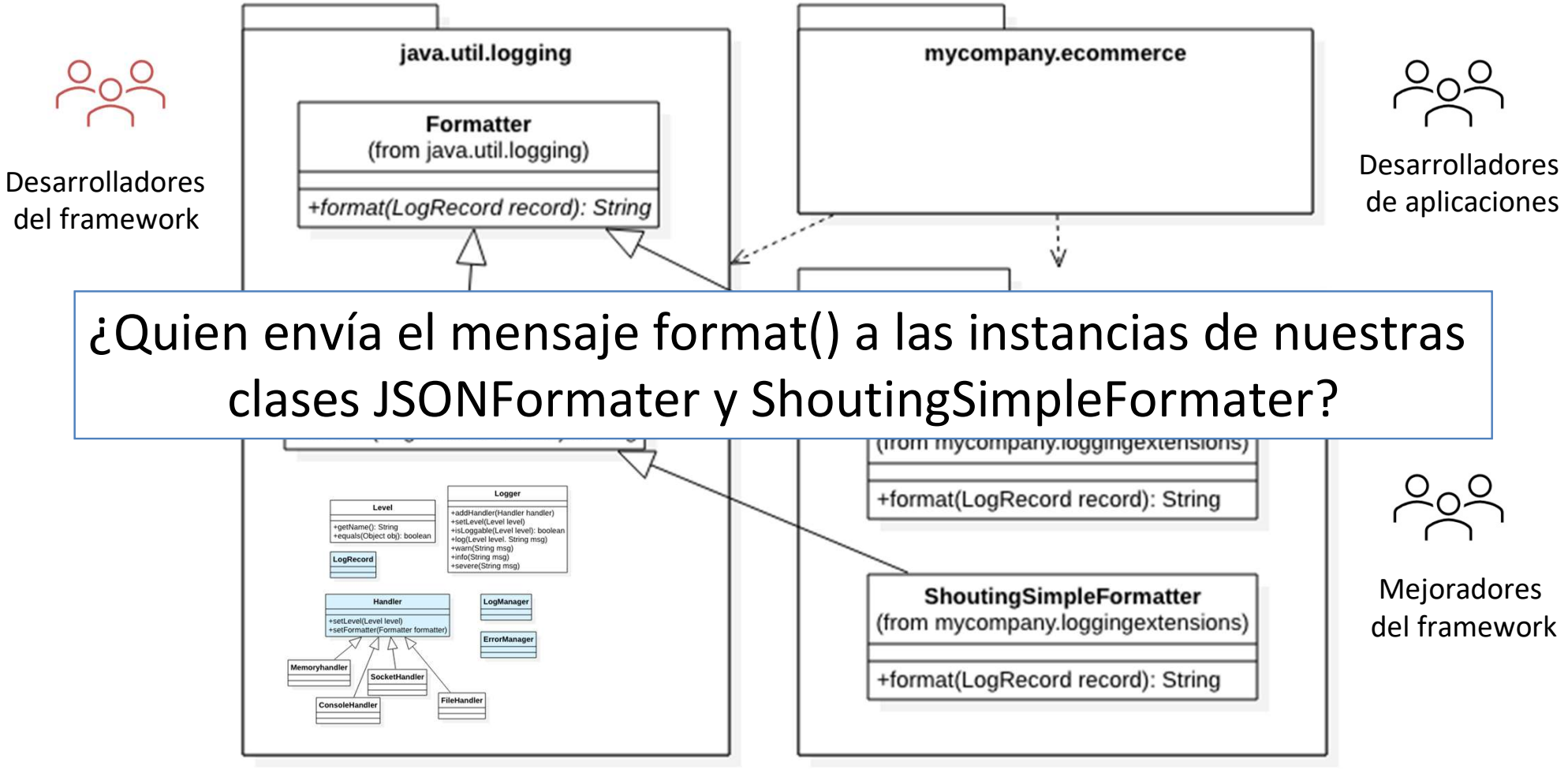
```java
public class JSONFormatter extends Formatter {
    @Override
    public String format(LogRecord record) {
        // Do whatever necesssary to represent record as
        // a JSON formatted string and return it
        return "...";
    }
}
```

Desarrolladores del framework

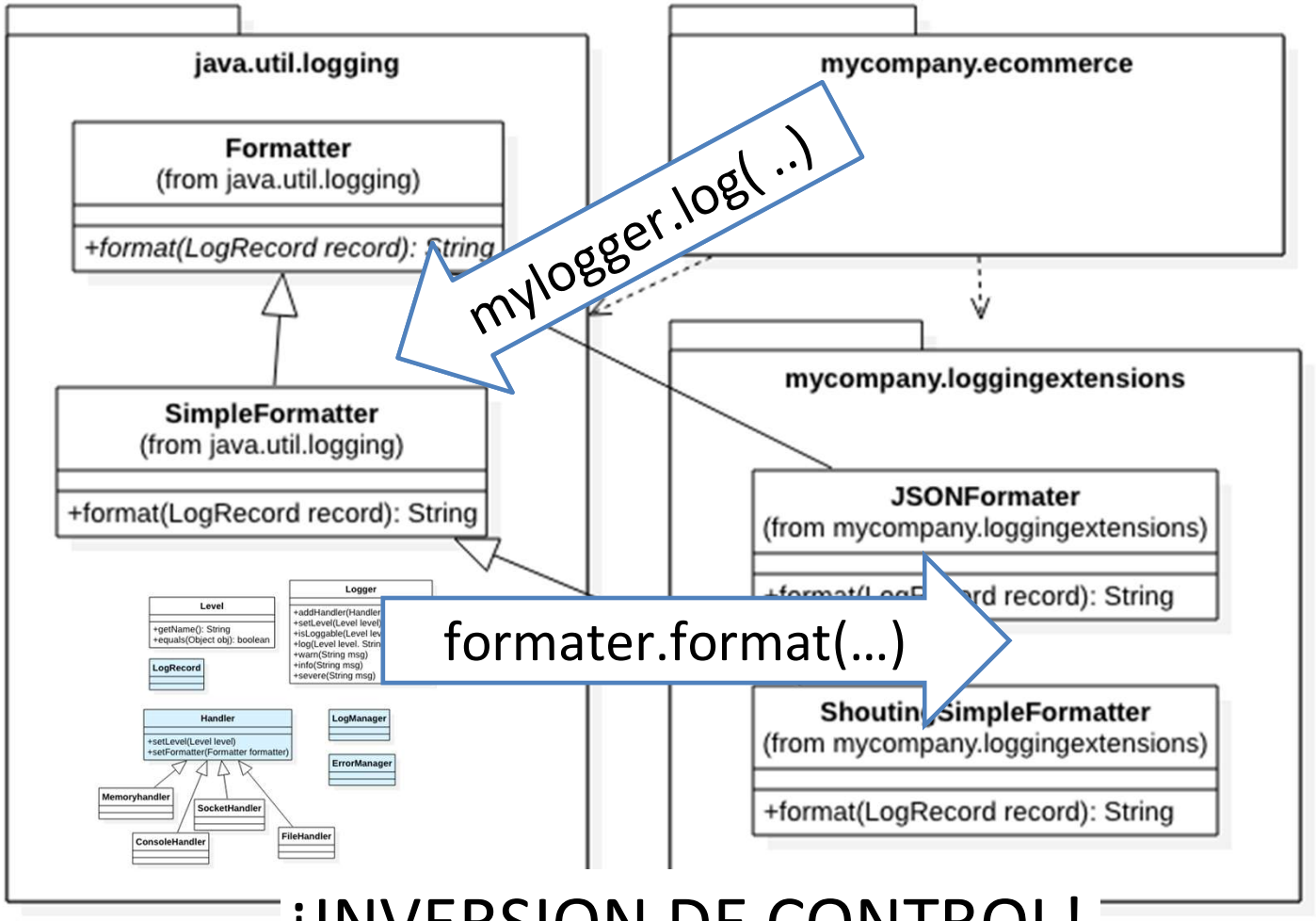


Desarrolladores de aplicaciones

Mejoradores del framework


Desarrolladores del framework
java.util.logging
Formatter
(from java.util.logging)
+format(LogRecord record): String
mycompany.ecommerce
Desarrolladores de aplicaciones
¿Quien envía el mensaje format() a las instancias de nuestras clases JSONFormater y ShoutingSimpleFormater?
Level
+getName(): String
+equals(Object obj): boolean
Logger
+addHandler(Handler handler)
+setLevel(Level level)
+isLoggable(Level level): boolean
+log(Level level. String msg)
+warn(String msg)
+info(String msg)
+severe(String msg)
LogRecord
Handler
+setLevel(Level level)
+setFormatter(Formatter formatter)
LogManager
ErrorManager
Memoryhandler
SocketHandler
ConsoleHandler
FileHandler
+format(LogRecord record): String
ShoutingSimpleFormatter
(from mycompany.loggingextensions)
+format(LogRecord record): String
Mejoradores del framework

Desarrolladores del framework



Desarrolladores de aplicaciones

Mejoradores del framework

¡INVERSION DE CONTROL!

Resumiendo

- java.util.logging es un framework que propone una forma de integrar logging en nuestros programas
- Lo utilizo mayormente como una caja negra (instancio y compongo)
- Toma el control cuando le indicamos, y lo devuelve cuando termina
- Puedo extender el framework heredando en los puntos de extensión previstos (caja blanca)
  - En este caso, vamos a percibir la inversión de control

Ejercicios para profundizar

- Armar un árbol de loggers explorando distintas alternativas
- Mirar la documentación y código fuente y reconstruir el diseño
- ¿Encotramos algún patrón en el diseño del framework?
- Escribir nuestro propio handler (¿enviar mensajes por telegram?)
- Escribir nuestro propio formatter (¿JSON?)
- Escribir nuestro propio filter (¿ignorar mensajes severos si no incluyen una excepción?)