Budapesti Műszaki és Gazdaságtudományi Egyetem

Java technológia

Logging

Sipos Róbert

siposr@hit.bme.hu

2014. 03. 20.

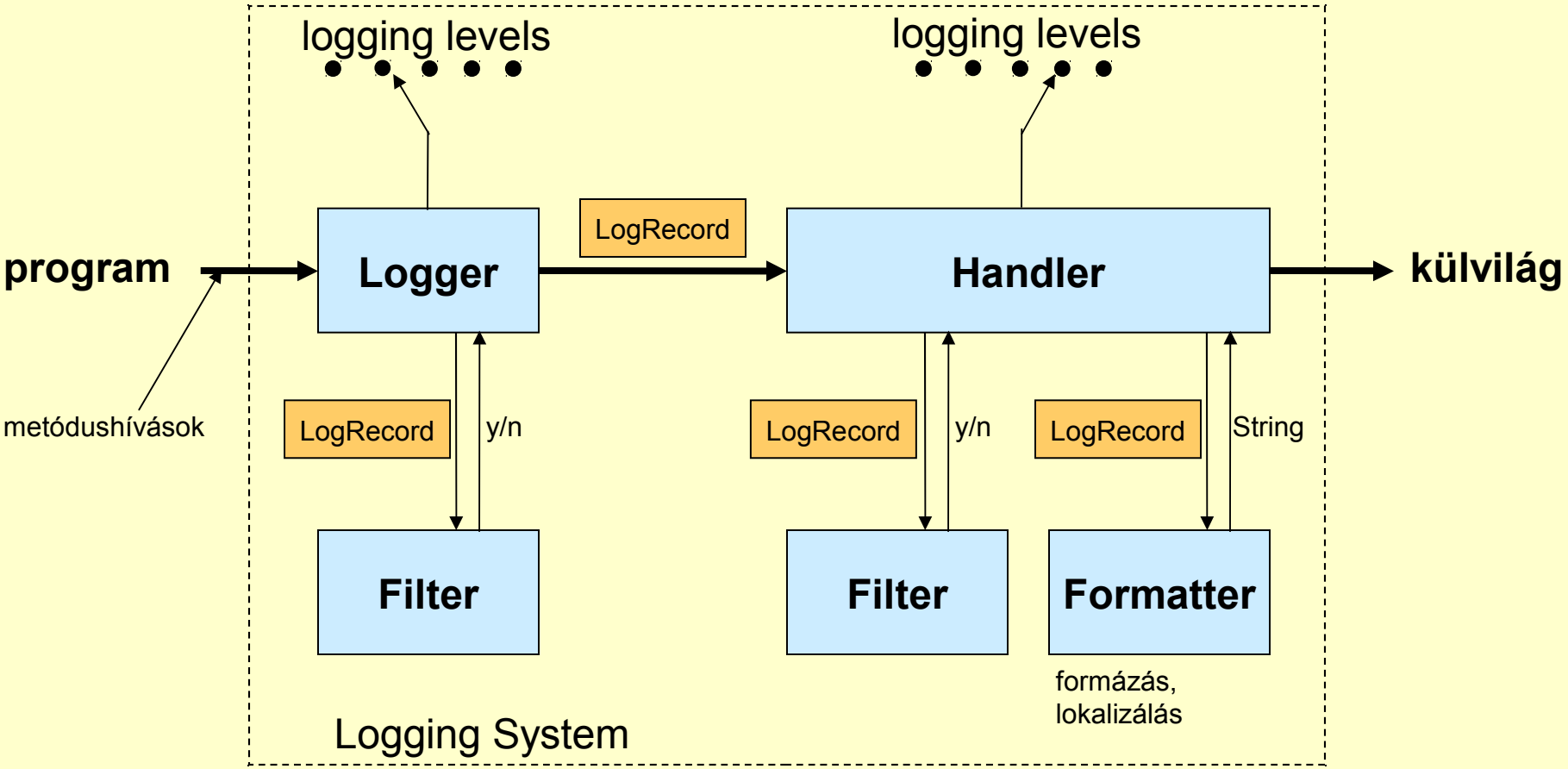
Bevezetés 1.

- Java 1.4 óta
- Logok használata:
  - Hibák, kivételek jelzése a végfelhasználónak vagy a rendszergazdának (konfigurációs hiba, erőforráshiány, biztonsági hiba ...)
  - Tesztelési fázisban a fellépő kivételek részletes leírása (+kontextus). Ezeket a logokat eljuttatni a fejlesztőkhöz.
  - A fejlesztési fázisban hibadetektálásra, végrehajtási történet követésére, események bekövetkeztének ellenőrzésére.

Bevezetés 2.

- A különféle felhasználások különféle részletezettséget követelnek meg:
  - A fellépő hiba/esemény pontos leírása (esetleg a kiváltó ok megjelölése), ember által kényelmesen olvasható formában. Nem tartalmazhat a rendszer belső felépítésére utaló információt.
  - A hiba és körülményeinek minél szabatosabb leírása, a kontextus megjelölése, a hiba keletkezésének helye (modul, esetleg programsor), kiváltó oka (hasznos a stack trace használata)
  - A fejlesztő által saját maga számára készített nyomkövetés (nem kell, hogy nagyon pontos legyen, az a lényeg, hogy akinek szól az megértse). Ezek később vagy kikerülhetnek a kódból, vagy pedig pontosítani kell őket.

Control Flow



Hierarchia 1.

- A Logger-ek fa struktúrába szervezhetők. Lehet egy Logger-nek szülő Logger-e.
- Minden Logger alapértelmezés szerint az általa előállított LogRecord-okat továbbadja a szülőjének is. Ez kikapcsolható.
- A Logger-ek hierarchiába szervezése hasznos lehet pl. ha egy szerteágazó program minden egyes moduljának van egy részletes log-ja és az egész programnak pedig egy egyszerű log-ja, ahol csak az esemény bekövetkeztét jelezzük. Ha valaki kíváncsi a részletekre, akkor megnézi a specifikus log-ot.
- Egy Logger objektumhoz több Handler is tartozhat (pl. egy a fájlba íráshoz, egy a konzolra való íráshoz).

Hierarchia 2.

- Bizonyos Handler objektumok láncba fűzhetők.
- Ilyen például a **MemoryHandler** amelyik egy puffert tart fenn a beérkező LogRecord-oknak.
  - ☞ Ha a buffer megtelik az új rekordok kitolják a régieket.
  - ☞ Egy meghatározott esemény bekövetkezte esetén a puffere tartalmát átadja a megadott cél Handler objektumnak.
    - ➢ a push metódus direkt meghívása
    - ➢ a beérkező LogRecord magasabb szintű mint a push level
    - ➢ leszármaztatva az osztályt, a log metódusban tetszőleges elv szerint dönthetjük el, hogy mikor hívjuk meg a push-t
  - ☞ Hasznos, ha finom felbontásban akarunk loggolni olyan adathordozóra ami gyakori hívogatása lelassítaná a programot

Hierarchia 3.

- Ha a Handler objektumok láncba vannak fűzve, akkor formázás csak a lánc utolsó eleménél történik (ez általában erőforrásigényes feladat, és előre felesleges lenne megcsinálni, ha az adott LogRecord nem biztos, hogy eléri a lánc végét).
- A Logger API megalkotásakor arra törekedtek, hogy egészen a konkrét loggolás megvalósításáig minimális legyen az erőforrásigénye. Tehát ha egy programban nagyon részletes loggolást valósítanak meg, de végül csak egy magasabb szintre konfigurálják be a loggoló rendszert, akkor az ne jelentsen nagy overhead-et.
  - ☞ log level ellenőrzése egy egyszerű összehasonlítás
  - ☞ filterek szűrőmetódusainak megírásánál is az egyszerűségre kell törekedni
  - ☞ azok a rekordok jutnak csak ki a külvilágba, amelyek a legmagasabb log level-en és az összes szűrőn átmennek

Loggolási szintek

- Minden egyes log üzenet rendelkezik egy megadott fontossági szinttel.
- A Level osztályban definiáltak hét alapértelmezett loggolási szintet
  - SEVERE (legmagasabb prioritású)
  - WARNING
  - INFO
  - CONFIG
  - FINE
  - FINER
  - FINEST (legalacsonyabb prioritású)
- Ezen felül van még egy OFF szint (a loggolás kikapcsolására) és egy ALL szint (minden üzenet engedélyezésére)

Logger 1.

- A programból a Logger-en keresztül érkeznek a loggolási kérések a loggoló rendszerhez
- Minden egyes Logger rendelkezik egy loggolási küszöbbel (log level), az ennél kisebb prioritású logging üzeneteket eldobja
- Általában minden Logger rendelkezik névvel.
  - ☞ A neveket pontokkal szeparált hierarchikus névtérben értelmezzük (pl. "java.util.logging": a Logging API Logger-e ☺).
  - ☞ Hasonlóan a package hierarchiához (célszerű a logger névhierarchiát a loggolt program package hierarchiájának megfeleltetni)
  - ☞ A LogManager kezeli a Logger-ek névterét
- Létrehozhatók anonymous Logger-ek is. Ezek nincsenek a névtérben.

Logger 2.

- A Logger-ek hierarchiáját csak a nevük hierarchiája szabja meg.
- A Logger-ek szüleiket a névtér vizsgálata alapján találják meg.
- Egy Logger őse a legközelebbi létező őse a névtérben.
  - ☞ pl. az "**a.b.c.d**" őse az "**a.b.c**",ha az létezik
- A root Logger-nek nincs szülője (a neve: "")
- Az anonymous logger-ek szülője a root logger.
- A Logger-ek a szülőjüktől különféle paramétereket örökölhetnek pl:
  - ☞ loggolási szint (ha nem adjuk meg, a szülőét örökli)
  - ☞ Handler-ek (minden Logger alapértelmezésben az üzeneteit továbbadja az ősének, így bizonyos értelemben örökli a Handler-eit)

Logger 3.

- A Logger biztosít metódusokat, a log üzenetek generálásához:
  - **`log`** metódusok családja: log level, üzenet (string) és esetleg paraméterek az üzenet kiegészítéseként
  - **`logp`** metódusok családja (log precise): meg kell adni a forrásosztály és metódus (ahol a loggolási igény keletkezett) nevét is
  - **`logrb`** metódusok családja: azonos a logp típusú metódusokkal, csak átveszi egy ResourceBundle nevét amit a log üzenetek lokalizálásához használ fel, de normális esetben a lokalizáció a Formatterben történik meg

Logger 4.

- a metódusok belépési és kilépési eseményéhez, kivételekhez:
  - entering: ENTRY szöveggel, FINER szinttel
  - exiting: RETURN szöveggel, FINER szinttel
  - throwing: THROW szöveggel, FINER szinttel
- metódusok, az alapértelmezett szintek szerint (a programozó kényelmét szolgálják: csak egy string-et (az üzenet) kell átadni):
  - severe
  - warning
  - info
  - ...

Logger 5.

- Azoknál a metódusoknál, ahol nem kell megadni a hívó metódus és osztály nevét a logging rendszer megpróbálja ezeket a paramétereket megállapítani.
- A logging rendszer csak best effort-ot igér
  - ☞ a hívó osztály és metódus nevét a stack trace alapján próbálja megállapítani, de ez nem mindig lehetséges mivel a Virtuális Gépek optimalizáló folyamatai adott esetben teljesen felszámolhatják azt, ezért a kapott értékek csak közelítőek, illetve rosszak is lehetnek

Handler

- A Logger-től kapott LogRecord-ot formázza (ha be van állítva Formatter), majd továbbítja a külvilág felé
- A Handler csak egy absztrakt osztály, a J2SE tartalmaz néhány alapvető megvalósítást:
  - `StreamHandler`: egy OutputStream-re írja a formázott üzeneteket
  - `ConsoleHandler`: a System.err-ra írja az üzeneteket
  - `FileHandler`: egy file-ba, vagy log file-ok forgó rendszerébe (pl: log0, log1, ... , log5 ha az egyik eléri a mérethatárt, továbblép a következőre) írja az üzeneteket
  - `SocketHandler`: egy megadott TCP kapcsolatra írja az üzeneteket
  - `MemoryHandler`: puffereli az üzeneteket (már volt)

Formatter

- A Handler-től kapott LogRecord-ot formázza és egy String-et ad vissza
- A Formatter is csak absztrakt osztály, a J2SE tartalmazza két alapvető megvalósítását
  - `SimpleFormatter`: a LogRecord alapján egy "emberi fogyasztásra" is alkalmas 1-2 soros üzenetet hoz létre
  - `XMLFormatter`: szabványos XML formába önti a LogRecord tartalmát
    - A DTD a Java Logging API dokumentáció függelékében megtalálható

LogManager 1.

- Globális objektum, ami a loggoló rendszert tartja kézben
  - kezeli a Logger-ek hierarchikus névterét
  - a logging rendszer konfigurációját végzi a konfigurációs file-ból olvasott paraméterek alapján
- A konfigurációs file szabványos java.util.Properties formátumú
- A VM indulásakor a LogManager beolvassa a konfigurációs file-t
- Az alkalmazás specifikus konfigurációs file-ja system property-k segítségével adható meg:
  - java.util.logging.config.class
  - java.util.logging.config.file

LogManager 2.

- Ha a java.util.logging.config.class segítségével adjuk meg, akkor a LogManager betölti a megadott osztályt, példányosítja, és az osztály konstruktorának kell beolvasnia a konfigurációt.
  Az osztály a beolvasott konfiguráció betöltésére a LogManager
  `readConfiguration(InputStream ins)`
  metódusát használja fel.
- ha a ...class paraméter nincs beállítva akkor a LogManager a java.util.logging.config.file system property-ben megadott file-ból beolvassa a konfigurációt (ez Properties formátumú)
- Ha egyik paraméter sincs beállítva, akkor a LogManager az alapértelmezett konfigurációt olvassa be a lib/logging.properties file-ból

Dinamikus konfigurálás

- A loggolási rendszer konfigurációja dinamikusan, futási időben is elvégezhető
  - ☞ létrehozhatók Logger-ek, megadhatók a Handler-ei
  - ☞ létrehozhatók és eltávolíthatók Handler-ek, Formatterek, Filterek
  - ☞ a loggolási szintek és egyéb paraméterek is változtathatók futási időben

Biztonság

- A logging rendszerben a biztonság alatt alapvetően azt értjük, hogy ismeretlen (untrusted) kód ne állíthassa el a logging rendszer konfigurációját, ezáltal meghiusítva a loggolást
- Ezért az appletek (untrusted) használhatják a névvel rendelkező Logger-eket, de nem változtathatnak a konfigurációs beállításokon
  - ☞ nem adhatnak hozzá, vagy vehetnek el Handler-eket
  - ☞ nem állíthatják a loggolási szinteket
- Az anonymous logger-ek privát loggerek, mivel a nevük alapján nem kérhetőek el (csak referenciával tudunk hivatkozni rájuk), ezek biztonsági ellenőrzése sokkal lazább (állítható a konfigurációjuk "untrusted" kód által is...)

Objektumok létrehozása

- Logger-ek létrehozása a Logger osztály statikus metódusaival:
  - Logger.getLogger(String name): ha már létezik a megadott nevű Logger, akkor azzal tér vissza, egyébként létrehoz egy új Logger-t az adott névvel
  - Logger.getAnonymousLogger() : létrehoz egy anonymous logger-t
    - általában Applet-ek használják, mert nincs beregisztrálva a LogManager-be, a metódusait nem védi a SecurityManager ezért az Applet-ek létrehozhatnak hozzá Handler-eket és Formatter-eket, és megváltoztathatják a konfigurációjukat

Példa 1.

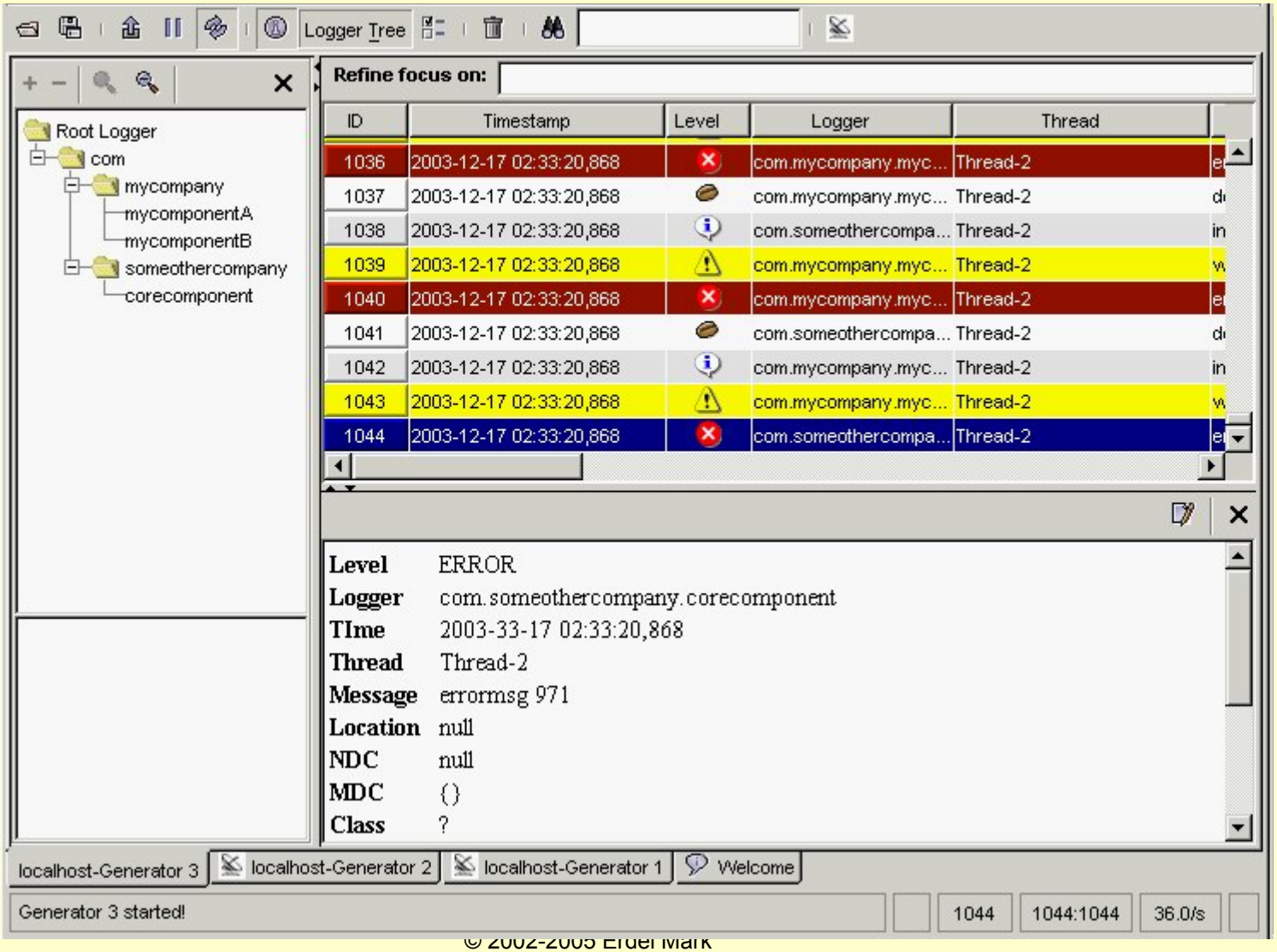
```
package org.bme.hit.javatargy.logging;

public class Pelda1 {

    private Logger logger;


    public Pelda1() {
       logger = Logger.getLogger(Pelda1.class.getName());
       logger.info("Sikerult loggolnom");
       helloWorld();
    }


    public void  helloWorld() {
       logger.entering("Pelda1","helloWorld");
       logger.info("Hello world!");
       logger.exiting("Pelda1","helloWorld");
    }

}
```

Példa XML kimenet

- Az XMLFormatter által generált kimenet:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2000-08-23 19:21:05</date>
  <millis>967083665789</millis>
  <sequence>1256</sequence>
  <logger>org.bme.hit.javatargy.logging.Pelda1</logger>
  <level>INFO</level>
  <class>Pelda1</class>
  <method>helloWorld</method>
  <thread>10</thread>
  <message>Hello world!</message>
</record>
</log>
```

Apache Chainsaw



Példa 2.

```
package org.bme.hit.javatargy.logging;

public class Pelda2 {

    public Pelda2() {
       logger = Logger.getLogger(Pelda1.class.getName());
       logger.setLevel(Level.FINEST);
       logger.addHandler(new FileHandler("%h/.log")); // user.home
       logger.info("Sikerult loggolnom");
       helloWorld();
    }
    public void  helloWorld() throws Exception {
       logger.entering("Pelda1","helloWorld");
       logger.info("Hello world!");
       Exception ex = new Exception("Hello world");
       logger.throwing("Pelda1","helloWorld",ex);
       throw ex;
    }
    private static Logger logger;

}
```