Ukládání konfigurace, internacionalizace a lokalizace, záznam činnosti aplikace

Obsah

Principy

Aplikace v globálním kontextu

- Rozsáhlé, mezinárodní, často dopředu neidentifikovatelné uživatelské skupiny
- Vhodná nastavení pro daný národní kontext (nejběžnějšího uživatele)
- Možnost personalizovatelnosti, zohlednění nastavení v systému a/nebo prohlížeči podle individuálních zvyklostí uživatele

Co internacionalizace řeší

Jednoduše, jde o to vzít do úvahy následující národní a kulturní odlišnosti a psát aplikace tak, aby se s nimi vyrovnaly:

- Různé abecedy, směry psaní písem
- Různorodé jazyky, mnohdy silné národní odlišnosti i v rámci jednotlivých jazyků
- Časová pásma, zobrazení času
- Rozdílná pojetí kalendářů
- Odlišné standardy zápisu čísel (desetinná místa, oddělovače řádů)
- Různé měny a způsoby zápisu částek v měně

Čím internacionalizujeme

Čísla a měna	pomocí třídy `NumberFormat`
Časové údaje	třídy `TimeZone`, `DateFormat`, `Calendar`
Abecední řazení	třída `Collator`, metoda `Arrays.sort`
Jazykové mutace řetězců	třídy `ResourceBundle` a `MessageFormat`

Co je lokalizace?

Internacionalizace (**i18n**) připraví půdu, aby bylo možné - a to i později - aplikaci pohodlně *lokalizovat* (**l10n**) do požadovaného národního prostředí.

- Dané národní prostředí je identifikováno a charakterizováno pomocí objektu `Locale`.
- Předáním tohoto objektu do metod se zajistí, že budou fungovat se zohledněním zvolených národních specifik.

Další zdroje

Wiki FI

- Téma: I18n/Internacionalizace [http://kore.fi.muni.cz:5080/wiki/index.php/I18n_-_Internacionalizace]

Zdroje Sun

Trail: Internationalization

- http://java.sun.com/docs/books/tutorial/i18n/

Principy

Záznam činnosti programu

- Při ladění k identifikaci chyby a zjištění její příčiny - jemná diagnostika
- V produkční fázi - záznam neobvyklých situací (vzniklých nejen vinou programu), případně monitoring zátěže, analýza uživatelských požadavků

Celkově

- **Java Logging API** je standardizací déletrvajících snah o jednotný přístup k záznamům (logování)
- Není řešením ideálním, hlavně proto, že přichází pozdě (`Log4J` a další tu již byly)
- Základní potřeby nicméně pokrývá zhruba stejně dobře
- Je součástí Java Core API

Princip fungování

- Ve zkoumaném/sledovaném kódu aktivujeme (vytvoříme, zpřístupníme) objekt třídy `Logger`
- Na inkriminovaných místech kódu voláme metody záznamu (`log`)

```
import java.util.logging.Logger;

private final static Logger LOGGER = Logger.getLogger(MyClass.class .getName());
```

Základní pojmy

Záznamník (LOGGER)

- Objekt vytvořený výše uvedeným způsobem (instance třídy `Logger`)
- Klíčová komponenta systému záznamů
- Přes něj jdou všechny žádosti o záznam - voláním metody `log`
- Záznamník má nastavenu *úroveň* (LEVEL) a je navázán na jeden či více *zpracovatelů* (HANDLER)

Úroveň záznamu (LEVEL)

- Záznamy jsou odlišeny podle závažnosti události do úrovní (LEVEL):
  - SEVERE (highest)
  - WARNING
  - INFO
  - CONFIG
  - FINE
  - FINER
  - FINEST (lowest)
- voláme různé metody objektu LOGGER
- na příslušném "loggeru" můžeme od dané úrovně níže záznamy zablokovat - nebudou vůbec zaznamenány

Zpracovatel záznamu (HANDLER)

- Na jednom záznamníku je navěšen jeden nebo více zpracovatelů zajišťujících fyzický výstup nebo uložení zprávy
- Základní jsou:

ConsoleHandler	Vypisuje zprávy na konzolu (chybový výstup)
FileHandler	Ukládá zprávy do souboru/ů

Formátovač výstupu (FORMATTER)

- Zpracovatel (HANDLER) má přiřazen tzv. formátovač definující přesný formát vypisovaných nebo ukládaných zpráv
- Základní jsou:

SimpleFormatter	Produkuje (plain)textový výstup
XMLFormatter	Produkuje XML na výstup

Příklad

Ukázka inicializace a volání metod záznamu

Převzato z H. Vogel: Java Logging API - Tutorial [http://www.vogella.de/articles/Logging/article.html].

```
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;

public class UseLogger {
 // Always use the classname, this way you can refactor
 private final static Logger LOGGER = Logger.getLogger(UseLogger.class
   .getName());

 public void writeLog() {
  // Set the LogLevel to Severe, only severe Messages will be written
  LOGGER.setLevel(Level.SEVERE);
  LOGGER.severe("Info Log");
  LOGGER.warning("Info Log");
  LOGGER.info("Info Log");
  LOGGER.finest("Really not important");

  // Set the LogLevel to Info, severe, warning and info will be written
  // Finest is still not written
  LOGGER.setLevel(Level.INFO);
  LOGGER.severe("Info Log");
  LOGGER.warning("Info Log");
  LOGGER.info("Info Log");
  LOGGER.finest("Really not important");
 }
```

```
 public static void main(String[] args) {
  UseLogger logger = new UseLogger();
  try {
   MyLogger.setup();
  } catch (IOException e) {
   e.printStackTrace();
   throw new RuntimeException("Problems with creating the log files");
  }
  logger.writeLog();
 }
}
```

Log4J

Log4J

- Klasické, jedno z prvních API pro logování
- Kvalitnější, ale nyní již ve stínu Java Logging API
- Používáno v řadě "legacy" aplikací a systémů
- Více na log4j.org

Výhody oproti Logging API

- sesterské projekty Log4Perl, Log4Net, Log4Cxx (c++), Log4CPlus, Log4PHP a Log4plsql

Commons Logging

Commons Logging

Projekt Apache zastřešující více stávajících (evt. budoucích) API pro logování:

- most mezi různými (stávajícími) API pro logování a kódem je používajícím
- vývojáři nemusejí pro jednoduché účely příliš řešit odlišnosti konkrétního API