JAVA

Odbočka: Návrhové vzory (Design patterns)

Návrhové vzory

- obecné řešení problému, které se využívá při návrhu počítačových programů (Wikipedia)

- Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software

- různé druhy
  - tvorba objektů (creational)
  - struktura (structural)
  - chování (behavioral)
  - ...

Singleton pattern

- „jedináček“
- pouze jedna instance od dané třídy

```
public class Singleton {

 private static final Singleton INSTANCE =
                                          new Singleton();

 private Singleton() {
 }

 public static Singleton getInstance() {
   return INSTANCE;
 }
}
```

Singleton pattern

- jiná implementace

```
public enum Singleton {
    INSTANCE;

    private EnumSingleton() {

    }
}
```

- použití
  - java.lang.Runtime
  - ...

Factory pattern

- vytváření nových objektů
- (statická) metoda vytvářející nové objekty
  - polymorfismus při vytváření objekt
- výhody
  - skrytí vytváření
  - plná kontrola nad typem a počtem instancí
  - …
- příklady
  - `static Integer valueOf(int i)`
  - `static <E> List<E> of(E... elements)`

Factory pattern (příklad)

```
public class Complex {
  public double real;
  public double imaginary;

  public static Complex fromCartesian(double real,
                                      double imaginary) {
    return new Complex(real, imaginary);
  }

  public static Complex fromPolar(double modulus,
                                  double angle) {
    return new Complex(modulus * Math.cos(angle),
                       modulus * Math.sin(angle));
  }

  private Complex(double real, double imaginary) {
    this.real = real;
    this.imaginary = imaginary;
  }
}
```

Factory pattern (příklad)

```
public static ImageReader
createImageReader(ImageInputStreamProcessor iisp) {
  if (iisp.isGIF()) {
    return new GifReader(iisp.getInputStream());
  } else if (iisp.isJPEG()) {
    return new JpegReader(iisp.getInputStream());
  } else {
    throw new IllegalArgumentException("Unknown
                                        image type.");
  }
}
```

Factory pattern

- nevýhoda
  - nelze vytvářet potomky (privátní konstruktor)
  - lze obejít protected konstruktorem
    - nebezpečné – lze ignorovat factory metodu

java.util.logging

Přehled

- API pro vytváření logů

- aplikace používá *Logger*
  - metody log()
- *Logger* vytváří *LogRecord* a předává ho *Handleru*
- *Handler* zajišťuje vypisování logů
  - na obrazovku, do souboru,...
- *Filter* – filtrování logovaných zpráv
- *Formatter* – formátování zpráv
- *LogManager* – přímo se (obvykle) nepoužívá
  - jeden globální objekt, spravuje loggery

Logger

- hierarchická struktura - strom
  - logger posílá zprávy i do předka
  - názvy *loggerů* by měly kopírovat hierarchii tříd
- několik úrovní zpráv
  - java.util.logging.Level
    - SEVER
    - WARNING
    - INFO
    - CONFIG
    - FINE
    - FINER
    - FINEST
  - lze určit, od které úrovně výše se budou zprávy logovat (nižší se budou zahazovat)

Handler

- několik předdefinovaných handlerů
  - Handler – abstraktní třída, ostatní od ní dědí
  - StreamHandler – do OutputStream
  - ConsoleHandler – do System.err
  - FileHandler – do souboru
    - jeden nebo "rotování" souboru
  - SocketHander – po síti
  - MemoryHadler – do bufferu
- vlastní *handler*
  - podědit od Handler

Formatter

- SimpleFormatter
  - text
  - "human-readable"
- XMLFormatter
  - xml

Logování

- metody na Logger
  - podle levelu
    - sever(String msg)
    - warning(String msg)
    - ...
  - obecné
    - log(Level l, String msg)
    - log(Level l, String msg, Object o)
    - log(Level l, String msg, Throwable t)
  - odkud logováno
    - logp(Level l, String sourceClass, String sourceMethod, String msg)
    - …
  - „líné“ logování
    - void log(Level level, Supplier<String> msgSupplier)
    - void severe(Supplier<String> msgSupplier)
    - …

Příklad

```
static Logger logger =
Logger.getLogger("cz.cuni.mff.java.logging.TestLog");
...
logger.info("doing stuff");
try{
  ...
} catch (Throwable ex){
  logger.log(Level.WARNING, "exception occured",ex);
}
logger.info("done");
```

„Externí“ konfigurace

- pomocí properties
  - java.util.logging.config.file
    - obvyklá struktura pro properties (jmeno=hodnota)
      - <logger>.hadlers = … seznam handlerů pro daný logger
      - <logger>.level = level pro daný logger
      - .....
      - bez úvodního jména – kořenový logger
  - java.util.logging.config.class
    - třída zodpovědná za načítání konfigurace
      - předchozí property pak nemusí mít žádný význam

System.Logger

- mnoho různých (externích) logovacích knihoven
  - Log4J, SLF4J...
- System.Logger System.getLogger(String name)
  - od Java 9
  - vrátí logger
    - záleží na „nastavení“, jaký se použije
- System.Logger
  - void log(System.Logger.Level level, String msg)
  - void log(System.Logger.Level level, Supplier<String> msgSupplier)
  - ...

java.util

Čas, datum

java.util.Date

- reprezentace času s přesností na milisekundy
  - od 1.1.1970
- většina metod je *deprecated*
  - od JDK1.1 nahrazeny třídou **Calendar**
- konstruktory
  - `Date()`
    - instance bude reprezentovat čas ve chvíli vytvoření objektu
  - `Data(long date)`
    - instance bude reprezentovat daný čas
- metody – v podstatě jen na porovnávání
  - `boolean after(Date d)`
  - `boolean before(Date d)`
  - `int compareTo(Date d)`
- ostatní metody jsou *deprecated*

java.util.Calendar

- abstraktní třída
- jediný ne-abstract potomek
  - GregorianCalendar
- statické atributy
  - co lze zjišťovat a nastavovat
    - YEAR, MONTH, DAY_OF_WEEK, DAY_OF_MONTH, HOUR, MINUTE, SECOND, AM_PM, ...
  - pro měsíce – JANUARY, FEBRUARY, ...
  - pro dny v týdnu – SUNDAY, MONDAY, ...
  - další – AM, PM, ...

java.util.Calendar: metody

- získání instance – statické metody
  - `getInstance()`
    - implicitní time zone
  - `getInstance(TimeZone tz)`
- získání/nastavení času
  - `Date getTime()`
  - `long getTimeInMillis()`
  - `void setTime(Date d)`
  - `void setTimeInMillis(long t)`
- porovnávání
  - `boolean before(Object when)`
  - `boolean after(Object when)`

java.util.Calendar: metody

- získávání jednotlivých položek
  - `int get(int field)`
  - př. int day = cal.get(Calendar.DAY_OF_MONTH)
- nastavování jednotlivých položek
  - `void set(int field, int value)`
  - př. cal.set(Calendar.MONTH, Calendar.SEPTEMBER)
  - výsledný čas v milisekundách se přepočítá až při volání get(), getTime(), getTimeInMillis()
- přidávání k položkám
  - `void add(int field, int delta)`
  - pokud je třeba, upraví se i ostatní položky
  - výsledný čas v milisekundách se přepočítá ihned
- přidávání k položkám bez zasahu do vyšších položek
  - `void roll(int field, int amount)`
  - `void roll(int field, boolean up)`

java.util.TimeZone

- reprezentace časového pásma
- bere v úvahu i letní/zimní čas
- získání timezone
  - `TimeZone getDefault()`
    - statická metoda
    - vrátí timezone nastavenou v systému
  - `TimeZone getTimeZone(String ID)`
    - vrátí požadovanou timezone
- možná ID
  - `String[] getAvailableIDs()`
  - statická metoda
- ID jsou tvaru
  - "America/Los_Angeles"
  - GMT +01:00

Java

java.time

java.time

- náhrada za `Calendar` od Java 8
  - `Calendar` není deprecated
- instance z java.time jsou obvykle nemodifikovatelné
  - na rozdíl od instancí `Calendar`

- `Instant`
  - okamžik na časové ose
  - vytvoření
    - `static Instant now()`
    - `static Instant ofEpochMilli(long milli)`
    - `static Instant parse(CharSequence text)`
  - metody
    - `plus...(...), minus...(...), …`
    - `int get(TemporalField field)`

java.time

- `Duration`
  - doba mezi dvěma okamžiky
  - př:
    - `Instant start = Instant.now();`
    - ...
    - `Instant end = Instant.now();`
    - `Duration duration = Duration.between(start, end);`
  - vytvoření
    - `static Duration ofDays(long days)`
    - `static Duration ofHours(long hours)`
    - `static Duration ofMinutes(long minutes)`
    - ...
  - metody
    - `long toDays()`
    - `long toHours()`
    - ...

java.time

- `LocalDate`
- `LocalTime`
- `LocalDateTime`
  - datum/čas bez informace o časové zóně
  - vytvoření
    - `(LocalDate | LocalTime | LocalDateTime).now()`
    - `LocalDate.of(int year, int month, int dayOfMonth)`
    - `...of(...)`
  - metody
    - `plus`, `minus`, `get`, ...
- `ZonedDateTime`
  - datum a čas s časovou zónou
  - zóna – `ZoneId`

java.util

Timer

Použití

- plánování úloh pro budoucí vykonání
  - jednou nebo opakovaně
- úloha = TimerTask
- všechny úlohy nastavené v jednom Timer objektu se vykonávají jedním vláknem
  - úloha by měla rychle skončit
- nastavení úlohy
  - `void schedule(TimerTask t, Date d)`
    - naplánuje úlohu na daný čas
  - `void schedule(TimerTask t, Date d, long period)`
    - naplánuje úlohu opakovaně
    - period – doba v milisekundách mezi opakovaným spuštěním

Použití

- nastavení úlohy (pokr.)
  - `void schedule(TimerTask t, long delay)`
    - naplánuje úlohu na dobu *aktuální čas* + *delay*
  - `void schedule(TimerTask t, long delay, long period)`
    - naplánuje úlohu opakovaně
    - period – doba v milisekundách mezi opakovaným spuštěním
  - `void scheduleAtFixedRate(TimerTask t, Date d, long period)`
  - `void scheduleAtFixedRate(TimerTask t, long delay, long period)`
    - naplánuje úlohu opakovaně
    - period – doba v milisekundách mezi opakovaným spuštěním relativně vzhledem k času prvního vykonaní

Použití

- metoda `void cancel()`
  - zruší timer
  - další naplánované úlohy se už neprovedou
  - aktuálně prováděná úloha se dokončí
  - lze volat opakovaně
    - další volání nedělají nic
- třída `TimerTask`
  - implementuje interface Runnable
  - abstraktní třída – nutno implementovat metodu run()
  - další metody
    - void cancel()
      - zruší úlohu
    - long scheduledExecutionTime()
      - čas nejbližšího dalšího spuštění

„Timer“ moderněji

```
ScheduledExecutorService scheduler =
                    Executors.newScheduledThreadPool(1);

Runnable task = new Runnable() {
  public void run() {
    ...
  }
};

scheduler.scheduleAtFixedRate(task, 0, 120, SECONDS);

...

scheduler.shutdown();
```

java.util

java.util.regex

java.util.regex

- regulární výrazy

- třídy Pattern a Matcher
- typické použití

```
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
boolean b = m.matches();
```

- Matcher
  - matches() – „matchuje“ celý řetězec
  - find() – hledá další podsekvenci, která „matchuje“ výraz

java.util.regex

- pozor na „speciální znaky“
  - např. reg-výraz matchující zpětné lomítko "\\\\"
  - "\Q.....\E"
    - zrušení významu pro spec. znaky

java.util

Localization

java.util.Locale

- reprezentuje geografický, politický nebo kulturní region
- určuje, jak vypisovat texty, čísla, měnu, čas,...
- vytváření
  - `Locale(String language)`
  - `Locale(String language, String country)`
  - `Locale(String language, String country, String variant)`
  - př. new Locale("cs", "CZ")
- `static Locale[] getAvailableLocales()`
  - vrátí všechny nainstalované *locales*
- static Locale getDefault()
  - vratí *locale* nastavený v systému

java.util.ResourceBundle

- obsahuje "lokalizované" objekty
  - např. řetězce
- *bundly* vždy patří do skupiny se společným základním jménem – př. MyResources
  - plné jméno *bundlu* = zákl. jméno + identifikace *locale*
  - př. MyResources_cs, MyResources_de, MyResources_de_CH
  - implicitní *bundle* – pouze se základním jménem
  - každý bundle ve skupině obsahuje stejné věci, ale "přeložené" pro daný *locale*
  - pokud *bundle* pro požadovaný *locale* není, použije se implicitní *bundle*

ResourceBundle: použití

- získání *bundlu*
  - `ResourceBundle.getBundle("MyResources")`
  - `ResourceBundle.getBundle("MyResources", currentLocale)`
- *bundle* obsahuje dvojice klíč/hodnota
  - klíče jsou pro všechny *locale* stejné, hodnota je jiná
- použití

```
ResourceBundle rs =
  ResourceBundle.getBundle("MyResources");
...
button1 = new Button(rs.getString("OkKey"));
button1 = new Button(rs.getString("CancelKey"));
```

ResourceBundle: použití

- klíče jsou vždy typu String
- hodnota je jakákoli
- získání objektu z *bundlu*
  - `String getString(String key)`
  - `String[] getStringArray(String key)`
  - `Object getObject(String key)`
    - př: `int[]`

```
ai=(int[])rs.getObject("intList");
```

- ResourceBundle – abstraktní třída
- dvě implementace
  - ListResourceBundle
  - PropertyResourceBundle

ListResourceBundle

- abstraktní třída
- potomci musí definovat metodu
  - `Object[][] getContents()`

```
public class MyResources extends ListResourceBundle {
      public Object[][] getContents() {return contents;}
      static final Object[][] contents = {
              {"OkKey", "OK"},
              {"CancelKey", "Cancel"},
      };
 }
public class MyResources_cs extends ListResourceBundle {
      public Object[][] getContents() {return contents;}
      static final Object[][] contents = {
              {"OkKey", "OK"},
              {"CancelKey", "Zrušit"},
      };
 }
```

PropertiesResourceBundle

- není abstraktní
- při použití se nevytváří žádná třída
- lokalizované řetězce jsou v souborech
- jméno souboru
  - základní jméno + locale + ".properties"
  - př. myresources.properties
    myresources_cs.properties
- získání bundlu
  - `ResourceBundle.getBundle("myresources")`
- formát souboru
  - `klíč=hodnota`
  - `# komentář do konce řádku`

Vlastní implementace

- potomek přímo od ResourceBundle
- předefinovat metody
  - Object handleGetObject(String key)
  - Enumeration getKeys()

```
public class MyResources extends ResourceBundle {
  public Object handleGetObject(String key) {
    if (key.equals("okKey")) return "Ok";
    if (key.equals("cancelKey")) return "Cancel";
    return null;
  }
}
public class MyResources_cs extends ResourceBundle {
  public Object handleGetObject(String key) {
    // nemusí definovat všechny klíče
    if (key.equals("cancelKey")) return "Zrušit";
    return null;
  }
}
```