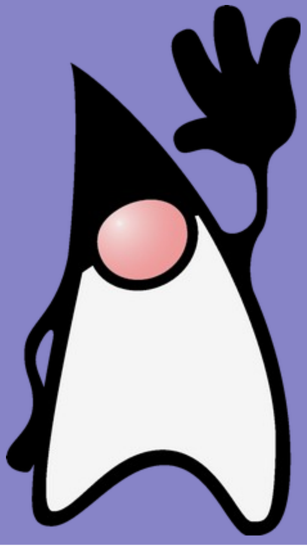
Java

Design patterns



Design patterns

- a general reusable solution to a commonly occurring problem within a given context in software design (Wikipedia)

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

- classification
  - creational
  - structural
  - behavioral
  - ...

Singleton pattern

- only a single instance of a given class

```
public class Singleton {

 private static final Singleton INSTANCE = new Singleton();

 private Singleton() {
 }

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

Singleton pattern

- another implementation

```
public enum Singleton{
    INSTANCE;

    private Singleton() {

    }
}
```

- usage
  - java.lang.Runtime
  - ...

Factory pattern

- creation of new objects

- a (static) method creating new objects
  - polymorphism during creation

- advantages
  - hiding creation
  - full control over types and number of instances
  - …

- examples
  - `static Integer valueOf(int i)`
  - `static <E> List<E> of(E... elements)`

Factory pattern (example)

```
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 (example)

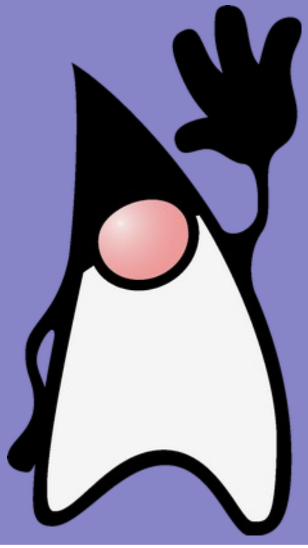
```
public static ImageReader
createImageReader(ImageInputStreamProcessor iisp) {
  if (iisp.isGIF()) {
    return new GifReader(iisp.getInputStream());
  } else if (iisp.isJPEG()) {
    return new JpegReader(iisp.getInputStream());
  } else {
    throw new IllegalArgumentException("Unknown image type.");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

Factory pattern

- disadvantage
  - cannot be extended (private constructor)
  - walk-around – protected constructor
    - dangerous – the factory method can be ignored

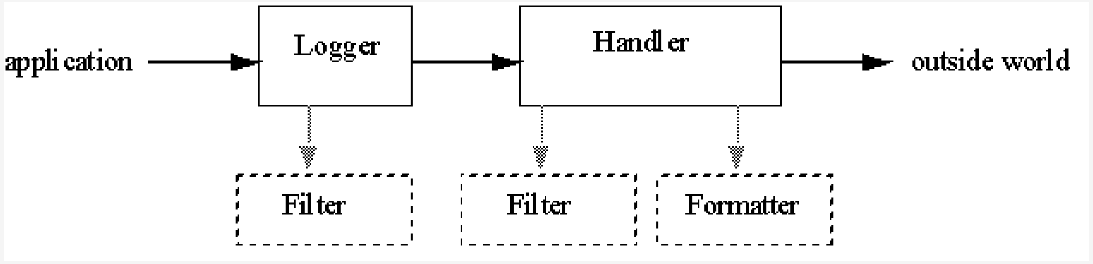
Java

java.util.logging



Overview

- API for logging



- an application uses the *Logger*
  - methods log()
- *Logger* creates *LogRecord* and passes it to *Handler*
- *Handler* prints out messages
  - on screen, to a file,...
- *Filter* – filtering logged messages
- *Formatter* – formating the messages
- *LogManager* – typically is not directly used
  - the single global object; manages the loggers

Logger

- hierarchical structure – tree
  - the logger sends messages also to the ancestor
  - names of the loggers should copy the hierarchy of classes
- several levels of messages
  - java.util.logging.Level
    - SEVER
    - WARNING
    - INFO
    - CONFIG
    - FINE
    - FINER
    - FINEST
  - it can be specified from which level the messages should be logged (messages with a lower level are ignored)

Handler

- several available handlers
  - Handler – the abstract class
    - other handlers extend it
  - StreamHandler – logs to an OutputStream
  - ConsoleHandler – to the System.err
  - FileHandler – to a file
    - to a single file or file “rotation”
  - SocketHander – to a socket
  - MemoryHadler – to a memory buffer
- own *handler*
  - extending the Handler

Formatter

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

Logging

- methods of the Logger
  - by the level
    - sever(String msg)
    - warning(String msg)
    - ...
  - generic ones
    - log(Level l, String msg)
    - log(Level l, String msg, Object o)
    - log(Level l, String msg, Throwable t)
  - with a logging source
    - logp(Level l, String sourceClass, String sourceMethod, String msg)
    - …
  - “lazy” logging
    - void log(Level level, Supplier<String> msgSupplier)
    - void severe(Supplier<String> msgSupplier)
    - ...

Example

```
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");
```

„External“ configuration

- using properties
  - java.util.logging.config.file
    - common format for properties (name=value)
      - <logger>.hadlers = … a list of handlers for the given logger
      - <logger>.level = a level for the given logger
      - .....
      - without the initial name – the root logger
  - java.util.logging.config.class
    - the class responsible for loading the configuration
      - the previous property then can have no meaning

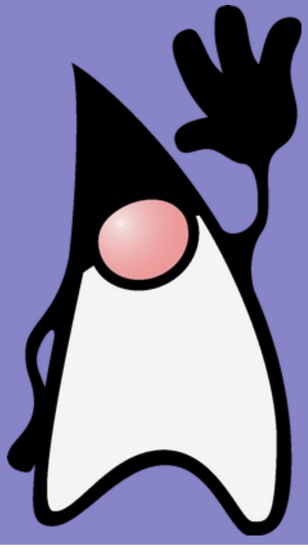
System.Logger

- many different (external) logging libraries
  - Log4J,…
- SLF4J – a bridging library (i.e., a wrapper around other logging libraries)

- System.Logger System.getLogger(String name)
  - since Java 9, also a “bridging library”
  - returns a logger
    - which one is used – depends on “configuration”
      - “configuration” via service loader – will be in NPRG021
    - by default – java.util.logging
- System.Logger
  - void log(System.Logger.Level level, String msg)
  - void log(System.Logger.Level level, Supplier<String> msgSupplier)
  - ...

Java

Time and date



java.util.Date

- represents time with millisecond precision
  - since 1.1.1970
- most of the methods are *deprecated*
  - since JDK1.1 replaced by **Calendar**
- constructors
  - `Date()`
    - an instance will hold time at which it was allocated
  - `Data(long date)`
    - an instance will hold the given time
- methods – in fact comparisons only
  - `boolean after(Date d)`
  - `boolean before(Date d)`
  - `int compareTo(Date d)`
- other ones are *deprecated*

java.util.Calendar

- abstract class
- the only non-abstract child
  - GregorianCalendar
- static attributs
  - what can be obtained/set
    - YEAR, MONTH, DAY_OF_WEEK, DAY_OF_MONTH, HOUR, MINUTE, SECOND, AM_PM, ...
  - months – JANUARY, FEBRUARY, ...
  - days in a week – SUNDAY, MONDAY, ...
  - other – AM, PM, ...

java.util.Calendar: methods

- obtaining an instance – static methods
  - `getInstance()`
    - default timezone
  - `getInstance(TimeZone tz)`
- getting/setting time
  - `Date getTime()`
  - `long getTimeInMillis()`
  - `void setTime(Date d)`
  - `void setTimeInMillis(long t)`
- comparison
  - `boolean before(Object when)`
  - `boolean after(Object when)`

java.util.Calendar: methods

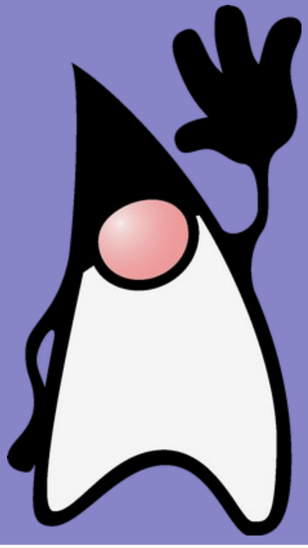
- obtaining individual fields
  - `int get(int field)`
  - ex. int day = cal.get(Calendar.DAY_OF_MONTH)
- setting individual fields
  - `void set(int field, int value)`
  - ex. cal.set(Calendar.MONTH, Calendar.SEPTEMBER)
  - resulting time in milliseconds is recalculated just during calls get(), getTime(), getTimeInMillis()
- adding to fields
  - `void add(int field, int delta)`
  - if necessary, modifies other fields also
  - resulting time in milliseconds is recalculated immediately
- adding to fields without modification of other fields
  - `void roll(int field, int amount)`
  - `void roll(int field, boolean up)`

java.util.TimeZone

- representation of a time zone
- understands summer/winter time
- obtaining a time zone
  - `TimeZone getDefault()`
    - static method
    - returns the timezone set in a system
  - `TimeZone getTimeZone(String ID)`
    - returns required time zone
- possible ID
  - `String[] getAvailableIDs()`
  - static method
- IDs have a form
  - "America/Los_Angeles"
  - GMT +01:00

Java

Date and Time – java.time



java.time

- since Java 8, replacement of `Calendar`
  - `Calendar` is not deprecated
- instances of java.time... are typically immutable
  - contrary to instances of `Calendar`

- `Instant`
  - an instantaneous point on the time-line
  - creation
    - `static Instant now()`
    - `static Instant ofEpochMilli(long milli)`
    - `static Instant parse(CharSequence text)`
  - methods
    - `plus...(...), minus...(...), …`
    - `int get(TemporalField field)`

java.time

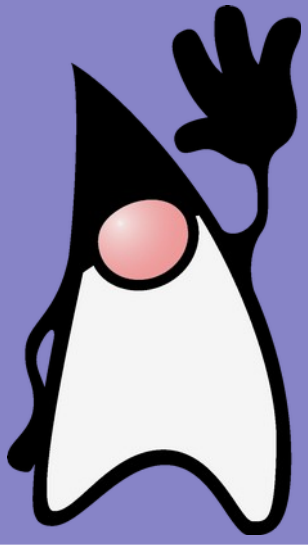
- `Duration`
  - amount of time between two time points
  - ex:
    - `Instant start = Instant.now();`
    - ...
    - `Instant end = Instant.now();`
    - `Duration duration = Duration.between(start, end);`
  - creation
    - `static Duration ofDays(long days)`
    - `static Duration ofHours(long hours)`
    - `static Duration ofMinutes(long minutes)`
    - `...`
  - methods
    - `long toDays()`
    - `long toHours()`
    - ...

java.time

- `LocalDate`
- `LocalTime`
- `LocalDateTime`
  - date/time without timezone
  - creation
    - (`LocalDate` | `LocalTime` | `LocalDateTime`).`now()`
    - `LocalDate.of(int year, int month, int dayOfMonth)`
    - `...of(...)`
  - methods
    - `plus`, `minus`, `get`, ...
- `ZonedDateTime`
  - date and time with timezone
  - zone – `ZoneId`

java.util

Timer



Usage

- scheduling tasks for future execution
  - one-time or repeated
- task = TimerTask
- all tasks in a single timer are executed in a single thread
  - a task should finish quickly
- scheduling a task
  - `void schedule(TimerTask t, Date d)`
    - schedules the task for the given time
  - `void schedule(TimerTask t, Date d, long period)`
    - schedules the task repeatedly
    - period – time in milliseconds between executions

Usage

- scheduling a task (cont.)
  - `void schedule(TimerTask t, long delay)`
    - schedules the task after given delay
  - `void schedule(TimerTask t, long delay, long period)`
    - schedules the task repeatedly
    - period – time in milliseconds between executions
  - `void scheduleAtFixedRate(TimerTask t, Date d, long period)`
  - `void scheduleAtFixedRate(TimerTask t, long delay, long period)`
    - schedules the task repeatedly
    - period – time in milliseconds between executions relatively to initial execution

Usage

- the method `void cancel()`
  - cancels the timer
  - no further scheduled tasks are executed
  - currently executed task is finished
  - can be called repeatedly
    - further calls do nothing
- the class `TimerTask`
  - implements the interface Runnable
  - abstract class – the run() method must be implemented
  - other methods
    - void cancel()
      - cancels the task
    - long scheduledExecutionTime()
      - time of the most recent actual execution

Modern “timer”

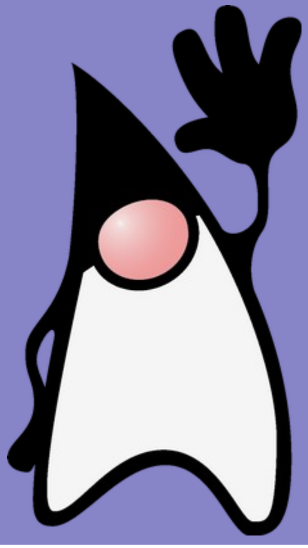
```
ScheduledExecutorService scheduler =
                        Executors.newScheduledThreadPool(1);

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

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

java.util

java.util.regex



java.util.regex

- regular expressions

- classes Pattern and Matcher
- typical usage

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

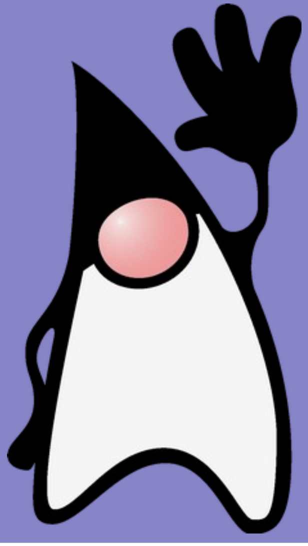
- Matcher
  - matches() – matches the entire string
  - find() – looking for the next subsequence that matches the pattern

java.util.regex

- warning - “special characters”
  - e.g. a regex matching the back-slash
    "\\\\"
  - "\Q......\E"
    - quoting all the characters in between

java.util

Localization



java.util.Locale

- represents a specific geographical, political, or cultural region
- defines how to print out texts, numbers, currency, time
- creation
  - `Locale(String language)`
  - `Locale(String language, String country)`
  - `Locale(String language, String country, String variant)`
  - ex. new Locale("cs", "CZ")
- `static Locale[] getAvailableLocales()`
  - returns all installed *locales*
- static Locale getDefault()
  - returns the default locale

java.util.ResourceBundle

- contains "localized" objects
  - e.g. strings
- *bundle* always bellongs to a group with common base name – e.g. MyResources
  - full name of a bundle = base name + locale id
  - ex. MyResources_cs, MyResources_de, MyResources_de_CH
  - default *bundle* – with the base name only
  - each bundle in a group holds the same objects transformed for a particular locale
  - if requested bundle is not available, the default one is used

ResourceBundle: Usage

- obtaining *bundels*
  - `ResourceBundle.getBundle("MyResources")`
  - `ResourceBundle.getBundle("MyResources", currentLocale)`
- *bundle* contains tuples key/value
  - keys are the same for oal locales in a group, the valueis different
- usage

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

ResourceBundle: Usage

- keys – String type
- value – any type
- obtaining an object from the buffer
  - `String getString(String key)`
  - `String[] getStringArray(String key)`
  - `Object getObject(String key)`
    - ex: `int[] ai=(int[])rs.getObject("intList");`
- ResourceBundle – abstract class
- two implementations
  - ListResourceBundle
  - PropertyResourceBundle

ListResourceBundle

- abstract class
- children must redefine the method
  - `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

- is not abstract
- no other class is directly created
- localized strings are in files
- a name of the file
  - base name + locale + ".properties"
  - ex. myresources.properties
    myresources_cs.properties
- obtaining the bundle
  - `ResourceBundle.getBundle("myresources")`
- the format of the file
  - `key=value`
  - `# comment till the end of the line`

Own implementation

- extending directly ResourceBundle
- overriding methods
  - 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) {
    // no need to define all keys
    if (key.equals("cancelKey")) return "Zrušit";
    return null;
  }
}
```