

## ExtendedProperties

2006-10-26 21:29:38

Wie ich bereits in meinem letzten Artikel schrieb, war es einfacher eine eigene kleine erweiterte Properties Klasse zu erstellen, als die 1 MB von Commons zu holen.

Hier nun der Code der ExtendedProperties, den man sicherlich noch um viele Getter erweitern könnte, die ich jedoch gerade nicht brauche.

```
import java.io.BufferedInputStream; import java.io.BufferedOutputStream; import java.io.File; import
java.io.FileInputStream; import java.io.FileOutputStream; import java.io.IOException; import
java.io.InputStream; import java.io.OutputStream; import java.net.MalformedURLException; import
java.net.URL; import java.text.SimpleDateFormat; import java.util.ArrayList; import java.util.Collections;
import java.util.Enumeration; import java.util.HashSet; import java.util.Iterator; import java.util.List; import
java.util.Map; import java.util.Properties; import java.util.Set; /** * This class extends the {@link
java.util.Properties} class with * additional functionality:<br/> * <dir> * <li>getters for Integer, int, Long,
long, Double, double, File, URL</li> * <li>getters for String arrays and Lists with configurable seperator
symbol</li> * <li>save and reload</li> * <li>auto save on finalization</li> * <li>setters for map and
properties</li> * </dir> * * @author <a href=&quot;http://narcanti.keyboardsamurais.de&quot;>M.
Serhat Cinar</a> * */ public class ExtendedProperties extends Properties{ /** * */ private static final
long serialVersionUID = 1L; private File source; /** * * @param source file where properties are
loaded/stored * @throws IOException */ public ExtendedProperties(File source) throws IOException{
super(); reload(); this.source = source; } /** * Uses the given Properties. * NEVER add a
ExtendedProperties here! * * @param props * @param source file where properties are loaded/stored
*/ public ExtendedProperties(Properties props, File source){ super(props); } /* (non-Javadoc) * @see
java.lang.Object#finalize() */ public void finalize(){ if (autoSaveOnFinalizer && isDirty){ try{ save(); }
catch(IOException e){ } } private boolean isDirty = false; private boolean autoSaveOnFinalizer = false;
/** * Returns if autosaving on finalizing is active. * Default false. * * @return * @see
#setAutoSaveOnFinalizer(boolean) */ public boolean isAutoSaveOnFinalizer() { return
autoSaveOnFinalizer; } /** * If set to true, the properties will be saved, when the {@link #finalize()}
method * is called on garbage-collection AND any value has changed ({@link #isDirty()} is true). *
Default false. * * @param autoSaveOnFinalizer */ public void setAutoSaveOnFinalizer(boolean
autoSaveOnFinalizer) { this.autoSaveOnFinalizer = autoSaveOnFinalizer; } /** * Returns if the properties
has changed. This property is set to true on any setter-methods * including {@link #clear()} and {@link
#reload()}. * and set to false on the {@link #save()}, {@link #store(OutputStream, String)} and {@link
#save(OutputStream, String)} * methods. * * @return */ public boolean isDirty() { return isDirty; } /** *
Returns the file where the properties are loaded/stored. * * @return */ public File getSourceFile(){ return
source; } /** * Saves the actual state of this properties. * * @return * @throws IOException */ public
ExtendedProperties save() throws IOException{ OutputStream out = null; try{ out = new
BufferedOutputStream(new FileOutputStream(source), 1024); store(out, &quot;# Last modified:
&quot;+SimpleDateFormat.getDateInstance(SimpleDateFormat.FULL, SimpleDateFormat.FULL));
out.flush(); } finally{ if (out != null){ try{ out.close(); } catch(Throwable t){ } } } // isDirty = false; return this; }
/* (non-Javadoc) * @see java.util.Properties#store(java.io.OutputStream, java.lang.String) */ public void
store(OutputStream out, String header) throws IOException{ super.store(out, header); isDirty = false; } /*
(non-Javadoc) * @see java.util.Properties#save(java.io.OutputStream, java.lang.String) */ public void
save(OutputStream out, String header){ super.save(out, header); isDirty = false; } /** * Reloads this
properties from the source file. * * @return * @throws IOException */ public ExtendedProperties
reload() throws IOException{ InputStream in = null; try{ in = new BufferedInputStream(new
```

```

FileInputStream(source), 1024); super.load(in); } finally{ if (in!=null){ try{ in.close(); } catch(Throwable t){ }
} } isDirty = false; return this; } // URL /** * Returns the given property as an url object. * * @param key
* @return given property as n file object or null if no property with given key was found */ public URL
getUrl(String key) throws MalformedURLException{ String prop = getProperty(key); if (isEmpty(prop, true))
return null; return new URL(prop.trim()); } // FILE /** * Returns the given property as a file object. * *
@param key * @return given property as a file object or null if no property with given key was found */
public File getFile(String key){ String prop = getProperty(key); if (isEmpty(prop, true)) return null; return
new File(prop.trim()); } // LONG, INT, DOUBLE, BOOLEAN /** * Returns the given property as a long.
* * @param key * @return given property as long or 0 if no property with given key was found */ public
long getLong(String key){ String prop = getProperty(key); if (isEmpty(prop, true)) return 0; return
Long.parseLong(prop.trim()); } /** * Returns the given property as a long object. * * @param key *
@return given property as a long object or null if no property with given key was found */ public Long
getLongL(String key){ String prop = getProperty(key); if (isEmpty(prop, true)) return null; return new
Long(prop.trim()); } /** * Returns the given property as an integer object. * * @param key * @return
given property as an integer object or null if no property with given key was found */ public Integer
getInteger(String key){ String prop = getProperty(key); if (isEmpty(prop, true)) return null; return new
Integer(prop.trim()); } /** * Returns the given property as an int. * * @param key * @return given
property as an int object or 0 if no property with given key was found */ public int getInt(String key){ String
prop = getProperty(key); if (isEmpty(prop, true)) return 0; return Integer.parseInt(prop.trim()); } /** *
Returns the given property as a double object. * * @param key * @return given property as a double
object or null if no property with given key was found */ public Double getDoubleD(String key){ String prop
= getProperty(key); if (isEmpty(prop, true)) return null; return new Double(prop.trim()); } /** * Returns the
given property as a double. * * @param key * @return given property as a double or 0 if no property
with given key was found */ public double getDouble(String key){ String prop = getProperty(key); if
(isEmpty(prop, true)) return 0; return Double.parseDouble(prop.trim()); } // STRING /** * Returns the
given property as a string. * * @param key * @return given property as a string or null if no property with
given key was found */ public String getString(String key){ return getProperty(key); } // BOOLEAN
private Set booleanTrueValues = new HashSet(); /** * Determines which values are representants of the
boolean value true when * using {@link #getBoolean(String)} and {@link #getBooleanB(String)}. * The
{@link Boolean#valueOf(java.lang.String)} is always used, but here some additional values, * like
"1" for true can be added. * All added values are internally stored after beeing trimmed and
lowercased, * so this is case insensitive. * * @param trueValues * @return */ public
ExtendedProperties setBooleanTrueValues(String [] trueValues){ booleanTrueValues.clear(); if
(trueValues!=null && trueValues.length>0) for (int i = 0; i<trueValues.length; i++)
booleanTrueValues.add(trueValues[i].trim().toLowerCase()); return this; } /** * Returns all values which
return true when using {@link #getBoolean(String)} and {@link #getBooleanB(String)}. * * @return *
@see #setBooleanTrueValues(String[]) */ public String [] getBooleanTrueValues(){ return (String [])
booleanTrueValues.toArray(new String[booleanTrueValues.size()]); } /** * If the given string matches
(trimmed and lowercased) any of * the values given by {@link #setBooleanTrueValues(String[])} or
makes {@link Boolean#valueOf(java.lang.String)} * return true, true is returned, false otherwise. * *
@param value * @return */ protected boolean isTrue(String value){ if (isEmpty(value, true)) return false;
if (booleanTrueValues.contains(value.trim().toLowerCase())) return true; return
Boolean.valueOf(value.trim().toLowerCase()).booleanValue(); } /** * Returns the given property as a
boolean object. * * @param key * @return given property as a boolean object or {@link
Boolean#FALSE} if no property with given key was found */ public Boolean getBooleanB(String key){ if
(isTrue(getProperty(key))) return Boolean.TRUE; return Boolean.FALSE; } /** * Returns the given
property as a boolean. * * @param key * @return given property as a boolean or false if no property
with given key was found */ public boolean getBoolean(String key){ return isTrue(getProperty(key)); } //
LIST /** * Default list seperator. */ public static final String DEFAULT_LISTSEPERATOR =
" , "; private String listSeperator = DEFAULT_LISTSEPERATOR; /** * Defines the
listseperator used for {@link #getArray(String)}, {@link #getList(String, boolean)} * and {@link

```

```

#getList(String, boolean, int)}. * Default is ','. * * @param seperator * @return */ public
ExtendedProperties setListSeperator(String seperator){ if (seperator!=null) this.listSeperator = seperator;
return this; } /** * Returns the actual listseperator used for {@link #getArray(String)}, {@link
#getList(String, boolean)} * and {@link #getList(String, boolean, int)}. * Default is ','. * * @return */
public String getListSeperator(){ return listSeperator; } /** * Returns the given property as an array of
strings. * The properties value is split with the actual listseperator. * * @param key * @return given
property as an array of strings or an array with size 0 if no property with given key was found * @see
#setListSeperator(String) * @see #getListSeperator() */ public String[] getArray(String key){ String prop =
getProperty(key); if (isEmpty(prop, true)) return null; return prop.split(listSeperator); } /** * Returns the
given property as a list of strings. * The properties value is split with the actual listseperator. * * @param
key * @param skipEmpty if true, empty listentries will be skipped and not added to the list * @return
given property as a list of strings or an empty list if no property with given key was found * @see
#setListSeperator(String) * @see #getListSeperator() */ public List getList(String key, boolean
skipEmpty){ String prop = getProperty(key); if (isEmpty(prop, true)) return Collections.EMPTY_LIST; String
[] values = prop.split(listSeperator); List props = new ArrayList(values.length); for (int i=0; i<values.length;
i++){ if (skipEmpty && isEmpty(values[i], true)) continue; props.add(values[i].trim()); } return props; } /** *
Constant for {@link #getList(String, boolean, int)} to create string objects */ public static final int
OBJECTTYPE_STRING = 0; /** * Constant for {@link #getList(String, boolean, int)} to create integer
objects */ public static final int OBJECTTYPE_INTEGER = 1; /** * Constant for {@link #getList(String,
boolean, int)} to create double objects */ public static final int OBJECTTYPE_DOUBLE = 2; /** *
Constant for {@link #getList(String, boolean, int)} to create long objects */ public static final int
OBJECTTYPE_LONG = 3; /** * Constant for {@link #getList(String, boolean, int)} to create boolean
objects */ public static final int OBJECTTYPE_BOOLEAN = 4; /** * Constant for {@link #getList(String,
boolean, int)} to create file objects */ public static final int OBJECTTYPE_FILE = 5; /** * Returns the
given property as a list of the defined objecttype. * The properties value is split with the actual
listseperator. * * @param key * @param skipEmpty if true, empty listentries will be skipped and not
added to the list * @return given property as a list of objects defined by objecttype or an empty list if no
property with given key was found * @see #setListSeperator(String) * @see #getListSeperator() * @see
#OBJECTTYPE_BOOLEAN * @see #OBJECTTYPE_DOUBLE * @see #OBJECTTYPE_FILE * @see
#OBJECTTYPE_INTEGER * @see #OBJECTTYPE_LONG * @see #OBJECTTYPE_STRING */ public
List getList(String key, boolean skipEmpty, int objectType){ String prop = getProperty(key); if
(isEmpty(prop, true)) return Collections.EMPTY_LIST; String [] values = prop.split(listSeperator); List
props = new ArrayList(values.length); for (int i=0; i<values.length; i++){ if (skipEmpty && isEmpty(values[i],
true)) continue; switch (objectType){ case OBJECTTYPE_STRING: default: props.add(values[i].trim());
break; case OBJECTTYPE_INTEGER: props.add(new Integer(values[i].trim())); break; case
OBJECTTYPE_DOUBLE: props.add(new Double(values[i].trim())); break; case OBJECTTYPE_LONG:
props.add(new Long(values[i].trim())); break; case OBJECTTYPE_BOOLEAN: if (isTrue(values[i].trim()))
props.add(Boolean.TRUE); else props.add(Boolean.FALSE); break; case OBJECTTYPE_FILE:
props.add(new File(values[i].trim())); break; } } return props; } // SETTERS /* (non-Javadoc) * @see
java.util.Map#clear() */ public void clear(){ isDirty = true; super.clear(); } /* (non-Javadoc) * @see
java.util.Dictionary#put(java.lang.Object, java.lang.Object) */ public Object put(Object key, Object value){
isDirty = true; return super.put(key, value); } /* (non-Javadoc) * @see
java.util.Dictionary#remove(java.lang.Object) */ public Object remove(Object key){ isDirty = true; return
super.remove(key); } /* (non-Javadoc) * @see java.util.Map#putAll(java.util.Map) */ public void
putAll(Map t){ isDirty = true; super.putAll(t); } /* (non-Javadoc) * @see
java.util.Properties#setProperty(java.lang.String, java.lang.String) */ public Object setProperty(String key,
String value){ isDirty = true; super.put(key, value); return super.setProperty(key, value); } /** * Sets the
given property using the {@link String#valueOf(java.lang.Object)} * method to determine the string
representation of the given value. * * @param key * @param value * @return */ public
ExtendedProperties setProperty(String key, Object value){ setProperty(key, String.valueOf(value)); return
this; } /** * Adds all entries from the given properties to this properties. * * @param props * @return */

```

# Narcanti

ExtendedProperties

```
public ExtendedProperties setProperties(Properties props){ Enumeration proptnames =
props.propertyNames(); String key; while (proptnames.hasMoreElements()){ key = (String)
proptnames.nextElement(); setProperty(key, props.getProperty(key)); } isDirty = true; return this; } /** *
Adds all entries from the given map to this properties. * * @param props * @return */ public
ExtendedProperties setProperties(Map props){ Iterator iter = props.keySet().iterator(); String key; while
(iter.hasNext()){ key = (String) iter.next(); setProperty(key, props.get(key)); } isDirty = true; return this; } //
UTIL /** * True if the given string is null or has length 0. * * @param s * @param trim if true, trims the
given string and returns true if length is 0 * @return */ public static final boolean isEmpty(String s,
boolean trim){ if (s==null || s.length()<=0) return true; if (trim && s.trim().length()<=0) return true; return
false; } }
```