Coverage Summary for Class: UrlValidator (<empty package name>)

Class	Class, %	Method, %	Line, %
UrlValidator	100% (1/ 1)	82.4% (14/ 17)	88.5% (92/ 104)

```
    1 /*
    2
  * Licensed to the Apache Software Foundation (ASF) under one or more
    3
  * contributor license agreements.  See the NOTICE file distributed with
    4
  * this work for additional information regarding copyright ownership.
    5
  * The ASF licenses this file to You under the Apache License, Version 2.0
    6
  * (the "License"); you may not use this file except in compliance with
    7  * the License.  You may obtain a copy of the License at
    8  *
    9  *      http://www.apache.org/licenses/LICENSE-2.0
   10  *
   11
  * Unless required by applicable law or agreed to in writing, software
   12  * distributed under the License is distributed on an "AS IS" BASIS,
   13
  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
   14
  * See the License for the specific language governing permissions and
   15  * limitations under the License.
   16  */
   17
   18
   19 import java.io.Serializable;
   20 import java.util.Arrays;
   21 import java.util.Collections;
   22 import java.util.HashSet;
   23 import java.util.Set;
   24 import java.util.regex.Matcher;
   25 import java.util.regex.Pattern;
   26
   27 /**
   28  * <p><b>URL Validation</b> routines.</p>
   29  * Behavior of validation is modified by passing in options:
   30
  * <li>ALLOW_2_SLASHES - [FALSE]  Allows double '/' characters in the path
   31  * component.</li>
   32
  * <li>NO_FRAGMENT- [FALSE]  By default fragments are allowed, if this option is
   33  * included then fragments are flagged as illegal.</li>
   34
  * <li>ALLOW_ALL_SCHEMES - [FALSE] By default only http, https, and ftp are
   35
  * considered valid schemes.  Enabling this option will let any scheme pass validati
   36  *
   37
  * <p>Originally based in on php script by Debbie Dyer, validation.php v1.2b, Date:
   38
```

```
 * http://javascript.internet.com. However, this validation now bears little resembl
  * to the php original.</p>
  * <pre>
  *   Example of usage:
  *
  *   Construct a UrlValidator with valid schemes of "http", and "https".
  *
  *    String[] schemes = {"http","https"}.
  *    UrlValidator urlValidator = new UrlValidator(schemes);
  *    if (urlValidator.isValid("ftp://foo.bar.com/")) {
  *       System.out.println("url is valid");
  *    } else {
  *       System.out.println("url is invalid");
  *    }
  *
  *    prints "url is invalid"
  *   If instead the default constructor is used.
  *
  *    UrlValidator urlValidator = new UrlValidator();
  *    if (urlValidator.isValid("ftp://foo.bar.com/")) {
  *       System.out.println("url is valid");
  *    } else {
  *       System.out.println("url is invalid");
  *    }
  *
  *   prints out "url is valid"
  *  </pre>
  *
  * @see
  * <a href="http://www.ietf.org/rfc/rfc2396.txt">
  *  Uniform Resource Identifiers (URI): Generic Syntax
  * </a>
  *
 * @version $Revision: 1227719 $ $Date: 2012-01-05 09:45:51 -0800 (Thu, 05 Jan 2012)
  * @since Validator 1.4
  */
 public class UrlValidator implements Serializable {


  private static final long serialVersionUID = 7557161713937335013L;

     /**
    * Allows all validly formatted schemes to pass validation instead of
      * supplying a set of valid schemes.
      */
     public static final long ALLOW_ALL_SCHEMES = 1 << 0;

     /**
      * Allow two slashes in the path component of the URL.
      */
     public static final long ALLOW_2_SLASHES = 1 << 1;

     /**
      * Enabling this options disallows any URL fragments.
      */
     public static final long NO_FRAGMENTS = 1 << 2;

     /**

```

```
     * Allow local URLs, such as http://localhost/ or http://machine/ .
 95
     * This enables a broad-brush check, for complex local machine name
 96
     *  validation requirements you should create your validator with
 97
     *  a {@link RegexValidator} instead ({@link #UrlValidator(RegexValidator, long)
 98       */
 99      public static final long ALLOW_LOCAL_URLS = 1 << 3;
100
101      // Drop numeric, and  "+-." for now
102
    private static final String AUTHORITY_CHARS_REGEX = "\\p{Alnum}\\-\\.";
103
104      /**
105       * This expression derived/taken from the BNF for URI (RFC2396).
106       */
107      private static final String URL_REGEX =
108
           "^(([^:/?#]+):)?(//([^/?#]*))?([^?#]*)(\\?([^#]*))?(#(.*))?";
109
    //                                                                  12

    private static final Pattern URL_PATTERN = Pattern.compile(URL_REGEX);
111
112      /**
113       * Schema/Protocol (ie. http:, ftp:, file:, etc).
114       */
115      private static final int PARSE_URL_SCHEME = 2;
116
117      /**
118       * Includes hostname/ip and port number.
119       */
120      private static final int PARSE_URL_AUTHORITY = 4;
121
122      private static final int PARSE_URL_PATH = 5;
123
124      private static final int PARSE_URL_QUERY = 7;
125
126      private static final int PARSE_URL_FRAGMENT = 9;
127
128      /**
129       * Protocol (ie. http:, ftp:,https:).
130       */
131
    private static final String SCHEME_REGEX = "^\\p{Alpha}[\\p{Alnum}\\+\\-\\.]*";

    private static final Pattern SCHEME_PATTERN = Pattern.compile(SCHEME_REGEX);
133
134      private static final String AUTHORITY_REGEX =
135              "^([" + AUTHORITY_CHARS_REGEX + "]*)(:\\d*)?(.*)?";
136
    //                                                                   1

    private static final Pattern AUTHORITY_PATTERN = Pattern.compile(AUTHORITY_REGEX
138
139      private static final int PARSE_AUTHORITY_HOST_IP = 1;
140
141      private static final int PARSE_AUTHORITY_PORT = 2;
142
143      /**
```

```
144         * Should always be empty.
145         */
146        private static final int PARSE_AUTHORITY_EXTRA = 3;
147
148
    private static final String PATH_REGEX = "^(/[-\\w:@&?=+,.!/~*'%$_;\\(\\)]*)?$";

    private static final Pattern PATH_PATTERN = Pattern.compile(PATH_REGEX);
150
151        private static final String QUERY_REGEX = "^(.*)$";
152
153
    private static final Pattern QUERY_PATTERN = Pattern.compile(QUERY_REGEX);
154
155        private static final String LEGAL_ASCII_REGEX = "^\\p{ASCII}+$";

    private static final Pattern ASCII_PATTERN = Pattern.compile(LEGAL_ASCII_REGEX);
157
158        private static final String PORT_REGEX = "^:(\\d{1,3})$";

    private static final Pattern PORT_PATTERN = Pattern.compile(PORT_REGEX);
160
161        /**
162         * Holds the set of current validation options.
163         */
164        private final long options;
165
166        /**
167         * The set of schemes that are allowed to be in a URL.
168         */
169        private final Set allowedSchemes;
170
171        /**
172
    * Regular expressions used to manually validate authorities if IANA
173         * domain name validation isn't desired.
174         */
175        private final RegexValidator authorityValidator;
176
177        /**
178         * If no schemes are provided, default to this set.
179         */

    private static final String[] DEFAULT_SCHEMES = {"http", "https", "ftp"};
181
182        /**
183
    * Singleton instance of this class with default schemes and options.
184         */

    private static final UrlValidator DEFAULT_URL_VALIDATOR = new UrlValidator();
186
187        /**
188
    * Returns the singleton instance of this class with default schemes and options
189         * @return singleton instance with default schemes and options
190         */
191        public static UrlValidator getInstance() {
            return DEFAULT_URL_VALIDATOR;
193        }
194
```

```
195      /**
196       * Create a UrlValidator with default properties.
197       */
198      public UrlValidator() {
             this(null);
200      }
201
202      /**
203
     * Behavior of validation is modified by passing in several strings options:
204
     * @param schemes Pass in one or more url schemes to consider valid, passing in
205       *        a null will default to "http,https,ftp" being valid.
206
     *        If a non-null schemes is specified then all valid schemes must
207
     *        be specified. Setting the ALLOW_ALL_SCHEMES option will
208       *        ignore the contents of schemes.
209       */
210      public UrlValidator(String[] schemes) {
             this(schemes, 0L);
212      }
213
214      /**
215       * Initialize a UrlValidator with the given validation options.
216
     * @param options The options should be set using the public constants declared
217
     * this class.  To set multiple options you simply add them together.  For examp
218       * ALLOW_2_SLASHES + NO_FRAGMENTS enables both of those options.
219       */
220      public UrlValidator(long options) {
             this(null, null, options);
222      }
223
224      /**
225       * Behavior of validation is modified by passing in options:
226       * @param schemes The set of valid schemes.
227
     * @param options The options should be set using the public constants declared
228
     * this class.  To set multiple options you simply add them together.  For examp
229       * ALLOW_2_SLASHES + NO_FRAGMENTS enables both of those options.
230       */
231      public UrlValidator(String[] schemes, long options) {
             this(schemes, null, options);
233      }
234
235      /**
236       * Initialize a UrlValidator with the given validation options.
237
     * @param authorityValidator Regular expression validator used to validate the a
238
     * @param options Validation options. Set using the public constants of this cla
239       * To set multiple options, simply add them together:
240       * <p><code>ALLOW_2_SLASHES + NO_FRAGMENTS</code></p>
241       * enables both of those options.
242       */
243
    public UrlValidator(RegexValidator authorityValidator, long options) {
             this(null, authorityValidator, options);
```

```
245      }
246
247      /**
248
     * Customizable constructor. Validation behavior is modifed by passing in option
249       * @param schemes the set of valid schemes
250
     * @param authorityValidator Regular expression validator used to validate the a
251
     * @param options Validation options. Set using the public constants of this cla
252       * To set multiple options, simply add them together:
253       * <p><code>ALLOW_2_SLASHES + NO_FRAGMENTS</code></p>
254       * enables both of those options.
255       */

     public UrlValidator(String[] schemes, RegexValidator authorityValidator, long op
              this.options = options;
258
              if (isOn(ALLOW_ALL_SCHEMES)) {
                  this.allowedSchemes = Collections.EMPTY_SET;
261           } else {
                  if (schemes == null) {
                      schemes = DEFAULT_SCHEMES;
264               }
                  this.allowedSchemes = new HashSet();
                  this.allowedSchemes.addAll(Arrays.asList(schemes));
267           }
268
              this.authorityValidator = authorityValidator;
270
271      }
272
273      /**
274       * <p>Checks if a field has a valid url address.</p>
275       *
276
     * @param value The value validation is being performed on.  A <code>null</code>
277       * value is considered invalid.
278       * @return true if the url is valid.
279       */
280      public boolean isValid(String value) {
              if (value == null) {
                  return false;
283           }
284
              if (!ASCII_PATTERN.matcher(value).matches()) {
                  return false;
287           }
288
289           // Check the whole url address structure
              Matcher urlMatcher = URL_PATTERN.matcher(value);
              if (!urlMatcher.matches()) {
                  return false;
293           }
294
              String scheme = urlMatcher.group(PARSE_URL_SCHEME);
              if (!isValidScheme(scheme)) {
                  return false;
298           }
299
              String authority = urlMatcher.group(PARSE_URL_AUTHORITY);
```

```
            if ("file".equals(scheme) && "".equals(authority)) {
302             // Special case - file: allows an empty authority
303         } else {
304             // Validate the authority
                if (!isValidAuthority(authority)) {
                    return false;
307             }
308         }
309
            if (!isValidPath(urlMatcher.group(PARSE_URL_PATH))) {
                return false;
312         }
313
            if (!isValidQuery(urlMatcher.group(PARSE_URL_QUERY))) {
                return false;
316         }
317
        if (!isValidFragment(urlMatcher.group(PARSE_URL_FRAGMENT))) {
319
                return false;
321
322         }
323
            return true;
325     }
326
327     /**
328      * Validate scheme. If schemes[] was initialized to a non null,
329
    * then only those scheme's are allowed.  Note this is slightly different
330      * than for the constructor.
331
    * @param scheme The scheme to validate.  A <code>null</code> value is considere
332      * invalid.
333      * @return true if valid.
334      */
335     protected boolean isValidScheme(String scheme) {
            if (scheme == null) {
                return false;
338         }
339
            if (!SCHEME_PATTERN.matcher(scheme).matches()) {
                return false;
342         }
343
            if (isOff(ALLOW_ALL_SCHEMES)) {
345
                if (!this.allowedSchemes.contains(scheme)) {
                    return false;
348             }
349         }
350
            return true;
352     }
353
354     /**
355
    * Returns true if the authority is properly formatted.  An authority is the com
356
    * of hostname and port.  A <code>null</code> authority value is considered inva
```

```
357          * @param authority Authority value to validate.
358          * @return true if authority (hostname and port) is valid.
359          */
360         protected boolean isValidAuthority(String authority) {
                if (authority == null) {
                    return false;
363             }
364
365             // check manual authority validation if specified
                if (authorityValidator != null) {
                    if (authorityValidator.isValid(authority)) {
                        return true;
369                 }
370             }
371
            Matcher authorityMatcher = AUTHORITY_PATTERN.matcher(authority);
                if (!authorityMatcher.matches()) {
                    return false;
375             }
376
            String hostLocation = authorityMatcher.group(PARSE_AUTHORITY_HOST_IP);
378             // check if authority is hostname or IP address:
379             // try a hostname first since that's much more likely
            DomainValidator domainValidator = DomainValidator.getInstance(isOn(ALLOW_LOC
                if (!domainValidator.isValid(hostLocation)) {
382                 // try an IP address
383                 InetAddressValidator inetAddressValidator =
                            InetAddressValidator.getInstance();
                    if (!inetAddressValidator.isValid(hostLocation)) {
386                     // isn't either one, so the URL is invalid
                        return false;
388                 }
389             }
390
                String port = authorityMatcher.group(PARSE_AUTHORITY_PORT);
                if (port != null) {
                    if (!PORT_PATTERN.matcher(port).matches()) {
                        return false;
395                 }
396             }
397
            String extra = authorityMatcher.group(PARSE_AUTHORITY_EXTRA);
                if (extra != null && extra.trim().length() > 0){
                    return false;
401             }
402
                return true;
404         }
405
406         /**
407
    * Returns true if the path is valid.  A <code>null</code> value is considered i
408          * @param path Path value to validate.
409          * @return true if path is valid.
410          */
411         protected boolean isValidPath(String path) {
                if (path == null) {
```

```
                return false;
            }

            if (!PATH_PATTERN.matcher(path).matches()) {
                return false;
            }

            int slash2Count = countToken("//", path);
            if (isOff(ALLOW_2_SLASHES) && (slash2Count > 0)) {
                return false;
            }

            int slashCount = countToken("/", path);
            int dot2Count = countToken("..", path);
            if (dot2Count > 0) {
                if ((slashCount - slash2Count - 1) <= dot2Count) {
                    return false;
                }
            }

            return true;
        }

        /**
     * Returns true if the query is null or it's a properly formatted query string.
         * @param query Query value to validate.
         * @return true if query is valid.
         */
        protected boolean isValidQuery(String query) {
            if (query == null) {
                return true;
            }

            return !QUERY_PATTERN.matcher(query).matches();
        }

        /**
     * Returns true if the given fragment is null or fragments are allowed.
         * @param fragment Fragment value to validate.
         * @return true if fragment is valid.
         */
        protected boolean isValidFragment(String fragment) {
            if (fragment == null) {
                return true;
            }

            return isOff(NO_FRAGMENTS);
        }

        /**
         * Returns the number of times the token appears in the target.
         * @param token Token value to be counted.
         * @param target Target value to count tokens in.
         * @return the number of tokens.
         */
        protected int countToken(String token, String target) {
            int tokenIndex = 0;
            int count = 0;
            while (tokenIndex != -1) {
```

```
                tokenIndex = target.indexOf(token, tokenIndex);
                if (tokenIndex > -1) {
                    tokenIndex++;
                    count++;
476             }
477         }
            return count;
479     }
480
481     /**
482
    * Tests whether the given flag is on.  If the flag is not a power of 2
483      * (ie. 3) this tests whether the combination of flags is on.
484      *
485      * @param flag Flag value to check.
486      *
487      * @return whether the specified flag value is on.
488      */
489     private boolean isOn(long flag) {
            return (this.options & flag) > 0;
491     }
492
493     /**
494
    * Tests whether the given flag is off.  If the flag is not a power of 2
495      * (ie. 3) this tests whether the combination of flags is off.
496      *
497      * @param flag Flag value to check.
498      *
499      * @return whether the specified flag value is off.
500      */
501     private boolean isOff(long flag) {
            return (this.options & flag) == 0;
503     }
504 }
```