001/\*  
002 \* Licensed to the Apache Software Foundation (ASF) under one or more  
003 \* contributor license agreements. See the NOTICE file distributed with  
004 \* this work for additional information regarding copyright ownership.  
005 \* The ASF licenses this file to You under the Apache License, Version 2.0  
006 \* (the "License"); you may not use this file except in compliance with  
007 \* the License. You may obtain a copy of the License at  
008 \*  
009 \* http://www.apache.org/licenses/LICENSE-2.0  
010 \*  
011 \* Unless required by applicable law or agreed to in writing, software  
012 \* distributed under the License is distributed on an "AS IS" BASIS,  
013 \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
014 \* See the License for the specific language governing permissions and  
015 \* limitations under the License.  
016 \*/  
017package org.apache.commons.beanutils;  
018  
019import java.beans.BeanInfo;  
020import java.beans.IntrospectionException;  
021import java.beans.Introspector;  
022import java.beans.PropertyDescriptor;  
023import java.lang.reflect.Constructor;  
024import java.lang.reflect.InvocationTargetException;  
025import java.lang.reflect.Method;  
026import java.util.AbstractMap;  
027import java.util.AbstractSet;  
028import java.util.ArrayList;  
029import java.util.Collection;  
030import java.util.Collections;  
031import java.util.HashMap;  
032import java.util.Iterator;  
033import java.util.Map;  
034import java.util.Set;  
035  
036import org.apache.commons.collections.Transformer;  
037import org.apache.commons.collections.keyvalue.AbstractMapEntry;  
038  
039/\*\*  
040 \* An implementation of Map for JavaBeans which uses introspection to  
041 \* get and put properties in the bean.  
042 \* <p>  
043 \* If an exception occurs during attempts to get or set a property then the  
044 \* property is considered non existent in the Map  
045 \*  
046 \* @version $Id$  
047 \*/  
048public class BeanMap extends AbstractMap<Object, Object> implements Cloneable {  
049  
050 private transient Object bean;  
051  
052 private transient HashMap<String, Method> readMethods = new HashMap<String, Method>();  
053 private transient HashMap<String, Method> writeMethods = new HashMap<String, Method>();  
054 private transient HashMap<String, Class<? extends Object>> types = new HashMap<String, Class<? extends Object>>();  
055  
056 /\*\*  
057 \* An empty array. Used to invoke accessors via reflection.  
058 \*/  
059 public static final Object[] NULL\_ARGUMENTS = {};  
060  
061 /\*\*  
062 \* Maps primitive Class types to transformers. The transformer  
063 \* transform strings into the appropriate primitive wrapper.  
064 \*  
065 \* N.B. private & unmodifiable replacement for the (public & static) defaultTransformers instance.  
066 \*/  
067 private static final Map<Class<? extends Object>, Transformer> typeTransformers =  
068 Collections.unmodifiableMap(createTypeTransformers());  
069  
070 /\*\*  
071 \* This HashMap has been made unmodifiable to prevent issues when  
072 \* loaded in a shared ClassLoader enviroment.  
073 \*  
074 \* @see "http://issues.apache.org/jira/browse/BEANUTILS-112"  
075 \* @deprecated Use {@link BeanMap#getTypeTransformer(Class)} method  
076 \*/  
077 @Deprecated  
078 public static HashMap defaultTransformers = new HashMap() {  
079 @Override  
080 public void clear() {  
081 throw new UnsupportedOperationException();  
082 }  
083 @Override  
084 public boolean containsKey(final Object key) {  
085 return typeTransformers.containsKey(key);  
086 }  
087 @Override  
088 public boolean containsValue(final Object value) {  
089 return typeTransformers.containsValue(value);  
090 }  
091 @Override  
092 public Set entrySet() {  
093 return typeTransformers.entrySet();  
094 }  
095 @Override  
096 public Object get(final Object key) {  
097 return typeTransformers.get(key);  
098 }  
099 @Override  
100 public boolean isEmpty() {  
101 return false;  
102 }  
103 @Override  
104 public Set keySet() {  
105 return typeTransformers.keySet();  
106 }  
107 @Override  
108 public Object put(final Object key, final Object value) {  
109 throw new UnsupportedOperationException();  
110 }  
111 @Override  
112 public void putAll(final Map m) {  
113 throw new UnsupportedOperationException();  
114 }  
115 @Override  
116 public Object remove(final Object key) {  
117 throw new UnsupportedOperationException();  
118 }  
119 @Override  
120 public int size() {  
121 return typeTransformers.size();  
122 }  
123 @Override  
124 public Collection values() {  
125 return typeTransformers.values();  
126 }  
127 };  
128  
129 private static Map<Class<? extends Object>, Transformer> createTypeTransformers() {  
130 final Map<Class<? extends Object>, Transformer> defaultTransformers =  
131 new HashMap<Class<? extends Object>, Transformer>();  
132 defaultTransformers.put(  
133 Boolean.TYPE,  
134 new Transformer() {  
135 public Object transform( final Object input ) {  
136 return Boolean.valueOf( input.toString() );  
137 }  
138 }  
139 );  
140 defaultTransformers.put(  
141 Character.TYPE,  
142 new Transformer() {  
143 public Object transform( final Object input ) {  
144 return new Character( input.toString().charAt( 0 ) );  
145 }  
146 }  
147 );  
148 defaultTransformers.put(  
149 Byte.TYPE,  
150 new Transformer() {  
151 public Object transform( final Object input ) {  
152 return Byte.valueOf( input.toString() );  
153 }  
154 }  
155 );  
156 defaultTransformers.put(  
157 Short.TYPE,  
158 new Transformer() {  
159 public Object transform( final Object input ) {  
160 return Short.valueOf( input.toString() );  
161 }  
162 }  
163 );  
164 defaultTransformers.put(  
165 Integer.TYPE,  
166 new Transformer() {  
167 public Object transform( final Object input ) {  
168 return Integer.valueOf( input.toString() );  
169 }  
170 }  
171 );  
172 defaultTransformers.put(  
173 Long.TYPE,  
174 new Transformer() {  
175 public Object transform( final Object input ) {  
176 return Long.valueOf( input.toString() );  
177 }  
178 }  
179 );  
180 defaultTransformers.put(  
181 Float.TYPE,  
182 new Transformer() {  
183 public Object transform( final Object input ) {  
184 return Float.valueOf( input.toString() );  
185 }  
186 }  
187 );  
188 defaultTransformers.put(  
189 Double.TYPE,  
190 new Transformer() {  
191 public Object transform( final Object input ) {  
192 return Double.valueOf( input.toString() );  
193 }  
194 }  
195 );  
196 return defaultTransformers;  
197 }  
198  
199  
200 // Constructors  
201 //-------------------------------------------------------------------------  
202  
203 /\*\*  
204 \* Constructs a new empty <code>BeanMap</code>.  
205 \*/  
206 public BeanMap() {  
207 }  
208  
209 /\*\*  
210 \* Constructs a new <code>BeanMap</code> that operates on the  
211 \* specified bean. If the given bean is <code>null</code>, then  
212 \* this map will be empty.  
213 \*  
214 \* @param bean the bean for this map to operate on  
215 \*/  
216 public BeanMap(final Object bean) {  
217 this.bean = bean;  
218 initialise();  
219 }  
220  
221 // Map interface  
222 //-------------------------------------------------------------------------  
223  
224 /\*\*  
225 \* Renders a string representation of this object.  
226 \* @return a <code>String</code> representation of this object  
227 \*/  
228 @Override  
229 public String toString() {  
230 return "BeanMap<" + String.valueOf(bean) + ">";  
231 }  
232  
233 /\*\*  
234 \* Clone this bean map using the following process:  
235 \*  
236 \* <ul>  
237 \* <li>If there is no underlying bean, return a cloned BeanMap without a  
238 \* bean.  
239 \*  
240 \* <li>Since there is an underlying bean, try to instantiate a new bean of  
241 \* the same type using Class.newInstance().  
242 \*  
243 \* <li>If the instantiation fails, throw a CloneNotSupportedException  
244 \*  
245 \* <li>Clone the bean map and set the newly instantiated bean as the  
246 \* underlying bean for the bean map.  
247 \*  
248 \* <li>Copy each property that is both readable and writable from the  
249 \* existing object to a cloned bean map.  
250 \*  
251 \* <li>If anything fails along the way, throw a  
252 \* CloneNotSupportedException.  
253 \*  
254 \* <ul>  
255 \*  
256 \* @return a cloned instance of this bean map  
257 \* @throws CloneNotSupportedException if the underlying bean  
258 \* cannot be cloned  
259 \*/  
260 @Override  
261 public Object clone() throws CloneNotSupportedException {  
262 final BeanMap newMap = (BeanMap)super.clone();  
263  
264 if(bean == null) {  
265 // no bean, just an empty bean map at the moment. return a newly  
266 // cloned and empty bean map.  
267 return newMap;  
268 }  
269  
270 Object newBean = null;  
271 final Class<? extends Object> beanClass = bean.getClass(); // Cannot throw Exception  
272 try {  
273 newBean = beanClass.newInstance();  
274 } catch (final Exception e) {  
275 // unable to instantiate  
276 final CloneNotSupportedException cnse = new CloneNotSupportedException  
277 ("Unable to instantiate the underlying bean \"" +  
278 beanClass.getName() + "\": " + e);  
279 BeanUtils.initCause(cnse, e);  
280 throw cnse;  
281 }  
282  
283 try {  
284 newMap.setBean(newBean);  
285 } catch (final Exception exception) {  
286 final CloneNotSupportedException cnse = new CloneNotSupportedException  
287 ("Unable to set bean in the cloned bean map: " +  
288 exception);  
289 BeanUtils.initCause(cnse, exception);  
290 throw cnse;  
291 }  
292  
293 try {  
294 // copy only properties that are readable and writable. If its  
295 // not readable, we can't get the value from the old map. If  
296 // its not writable, we can't write a value into the new map.  
297 final Iterator<?> readableKeys = readMethods.keySet().iterator();  
298 while(readableKeys.hasNext()) {  
299 final Object key = readableKeys.next();  
300 if(getWriteMethod(key) != null) {  
301 newMap.put(key, get(key));  
302 }  
303 }  
304 } catch (final Exception exception) {  
305 final CloneNotSupportedException cnse = new CloneNotSupportedException  
306 ("Unable to copy bean values to cloned bean map: " +  
307 exception);  
308 BeanUtils.initCause(cnse, exception);  
309 throw cnse;  
310 }  
311  
312 return newMap;  
313 }  
314  
315 /\*\*  
316 \* Puts all of the writable properties from the given BeanMap into this  
317 \* BeanMap. Read-only and Write-only properties will be ignored.  
318 \*  
319 \* @param map the BeanMap whose properties to put  
320 \*/  
321 public void putAllWriteable(final BeanMap map) {  
322 final Iterator<?> readableKeys = map.readMethods.keySet().iterator();  
323 while (readableKeys.hasNext()) {  
324 final Object key = readableKeys.next();  
325 if (getWriteMethod(key) != null) {  
326 this.put(key, map.get(key));  
327 }  
328 }  
329 }  
330  
331  
332 /\*\*  
333 \* This method reinitializes the bean map to have default values for the  
334 \* bean's properties. This is accomplished by constructing a new instance  
335 \* of the bean which the map uses as its underlying data source. This  
336 \* behavior for <code>clear()</code> differs from the Map contract in that  
337 \* the mappings are not actually removed from the map (the mappings for a  
338 \* BeanMap are fixed).  
339 \*/  
340 @Override  
341 public void clear() {  
342 if(bean == null) {  
343 return;  
344 }  
345  
346 Class<? extends Object> beanClass = null;  
347 try {  
348 beanClass = bean.getClass();  
349 bean = beanClass.newInstance();  
350 }  
351 catch (final Exception e) {  
352 final UnsupportedOperationException uoe =  
353 new UnsupportedOperationException("Could not create new instance of class: " + beanClass);  
354 BeanUtils.initCause(uoe, e);  
355 throw uoe;  
356 }  
357 }  
358  
359 /\*\*  
360 \* Returns true if the bean defines a property with the given name.  
361 \* <p>  
362 \* The given name must be a <code>String</code>; if not, this method  
363 \* returns false. This method will also return false if the bean  
364 \* does not define a property with that name.  
365 \* <p>  
366 \* Write-only properties will not be matched as the test operates against  
367 \* property read methods.  
368 \*  
369 \* @param name the name of the property to check  
370 \* @return false if the given name is null or is not a <code>String</code>;  
371 \* false if the bean does not define a property with that name; or  
372 \* true if the bean does define a property with that name  
373 \*/  
374 @Override  
375 public boolean containsKey(final Object name) {  
376 final Method method = getReadMethod(name);  
377 return method != null;  
378 }  
379  
380 /\*\*  
381 \* Returns true if the bean defines a property whose current value is  
382 \* the given object.  
383 \*  
384 \* @param value the value to check  
385 \* @return false true if the bean has at least one property whose  
386 \* current value is that object, false otherwise  
387 \*/  
388 @Override  
389 public boolean containsValue(final Object value) {  
390 // use default implementation  
391 return super.containsValue(value);  
392 }  
393  
394 /\*\*  
395 \* Returns the value of the bean's property with the given name.  
396 \* <p>  
397 \* The given name must be a {@link String} and must not be  
398 \* null; otherwise, this method returns <code>null</code>.  
399 \* If the bean defines a property with the given name, the value of  
400 \* that property is returned. Otherwise, <code>null</code> is  
401 \* returned.  
402 \* <p>  
403 \* Write-only properties will not be matched as the test operates against  
404 \* property read methods.  
405 \*  
406 \* @param name the name of the property whose value to return  
407 \* @return the value of the property with that name  
408 \*/  
409 @Override  
410 public Object get(final Object name) {  
411 if ( bean != null ) {  
412 final Method method = getReadMethod( name );  
413 if ( method != null ) {  
414 try {  
415 return method.invoke( bean, NULL\_ARGUMENTS );  
416 }  
417 catch ( final IllegalAccessException e ) {  
418 logWarn( e );  
419 }  
420 catch ( final IllegalArgumentException e ) {  
421 logWarn( e );  
422 }  
423 catch ( final InvocationTargetException e ) {  
424 logWarn( e );  
425 }  
426 catch ( final NullPointerException e ) {  
427 logWarn( e );  
428 }  
429 }  
430 }  
431 return null;  
432 }  
433  
434 /\*\*  
435 \* Sets the bean property with the given name to the given value.  
436 \*  
437 \* @param name the name of the property to set  
438 \* @param value the value to set that property to  
439 \* @return the previous value of that property  
440 \* @throws IllegalArgumentException if the given name is null;  
441 \* if the given name is not a {@link String}; if the bean doesn't  
442 \* define a property with that name; or if the bean property with  
443 \* that name is read-only  
444 \* @throws ClassCastException if an error occurs creating the method args  
445 \*/  
446 @Override  
447 public Object put(final Object name, final Object value) throws IllegalArgumentException, ClassCastException {  
448 if ( bean != null ) {  
449 final Object oldValue = get( name );  
450 final Method method = getWriteMethod( name );  
451 if ( method == null ) {  
452 throw new IllegalArgumentException( "The bean of type: "+  
453 bean.getClass().getName() + " has no property called: " + name );  
454 }  
455 try {  
456 final Object[] arguments = createWriteMethodArguments( method, value );  
457 method.invoke( bean, arguments );  
458  
459 final Object newValue = get( name );  
460 firePropertyChange( name, oldValue, newValue );  
461 }  
462 catch ( final InvocationTargetException e ) {  
463 final IllegalArgumentException iae = new IllegalArgumentException(e.getMessage());  
464 if (BeanUtils.initCause(iae, e) == false) {  
465 logInfo(e);  
466 }  
467 throw iae;  
468 }  
469 catch ( final IllegalAccessException e ) {  
470 final IllegalArgumentException iae = new IllegalArgumentException(e.getMessage());  
471 if (BeanUtils.initCause(iae, e) == false) {  
472 logInfo(e);  
473 }  
474 throw iae;  
475 }  
476 return oldValue;  
477 }  
478 return null;  
479 }  
480  
481 /\*\*  
482 \* Returns the number of properties defined by the bean.  
483 \*  
484 \* @return the number of properties defined by the bean  
485 \*/  
486 @Override  
487 public int size() {  
488 return readMethods.size();  
489 }  
490  
491  
492 /\*\*  
493 \* Get the keys for this BeanMap.  
494 \* <p>  
495 \* Write-only properties are <b>not</b> included in the returned set of  
496 \* property names, although it is possible to set their value and to get  
497 \* their type.  
498 \*  
499 \* @return BeanMap keys. The Set returned by this method is not  
500 \* modifiable.  
501 \*/  
502 @SuppressWarnings({ "unchecked", "rawtypes" })  
503 // The set actually contains strings; however, because it cannot be  
504 // modified there is no danger in selling it as Set<Object>  
505 @Override  
506 public Set<Object> keySet() {  
507 return Collections.unmodifiableSet((Set) readMethods.keySet());  
508 }  
509  
510 /\*\*  
511 \* Gets a Set of MapEntry objects that are the mappings for this BeanMap.  
512 \* <p>  
513 \* Each MapEntry can be set but not removed.  
514 \*  
515 \* @return the unmodifiable set of mappings  
516 \*/  
517 @Override  
518 public Set<Map.Entry<Object, Object>> entrySet() {  
519 return Collections.unmodifiableSet(new AbstractSet<Map.Entry<Object, Object>>() {  
520 @Override  
521 public Iterator<Map.Entry<Object, Object>> iterator() {  
522 return entryIterator();  
523 }  
524 @Override  
525 public int size() {  
526 return BeanMap.this.readMethods.size();  
527 }  
528 });  
529 }  
530  
531 /\*\*  
532 \* Returns the values for the BeanMap.  
533 \*  
534 \* @return values for the BeanMap. The returned collection is not  
535 \* modifiable.  
536 \*/  
537 @Override  
538 public Collection<Object> values() {  
539 final ArrayList<Object> answer = new ArrayList<Object>( readMethods.size() );  
540 for ( final Iterator<Object> iter = valueIterator(); iter.hasNext(); ) {  
541 answer.add( iter.next() );  
542 }  
543 return Collections.unmodifiableList(answer);  
544 }  
545  
546  
547 // Helper methods  
548 //-------------------------------------------------------------------------  
549  
550 /\*\*  
551 \* Returns the type of the property with the given name.  
552 \*  
553 \* @param name the name of the property  
554 \* @return the type of the property, or <code>null</code> if no such  
555 \* property exists  
556 \*/  
557 public Class<?> getType(final String name) {  
558 return types.get( name );  
559 }  
560  
561 /\*\*  
562 \* Convenience method for getting an iterator over the keys.  
563 \* <p>  
564 \* Write-only properties will not be returned in the iterator.  
565 \*  
566 \* @return an iterator over the keys  
567 \*/  
568 public Iterator<String> keyIterator() {  
569 return readMethods.keySet().iterator();  
570 }  
571  
572 /\*\*  
573 \* Convenience method for getting an iterator over the values.  
574 \*  
575 \* @return an iterator over the values  
576 \*/  
577 public Iterator<Object> valueIterator() {  
578 final Iterator<?> iter = keyIterator();  
579 return new Iterator<Object>() {  
580 public boolean hasNext() {  
581 return iter.hasNext();  
582 }  
583 public Object next() {  
584 final Object key = iter.next();  
585 return get(key);  
586 }  
587 public void remove() {  
588 throw new UnsupportedOperationException( "remove() not supported for BeanMap" );  
589 }  
590 };  
591 }  
592  
593 /\*\*  
594 \* Convenience method for getting an iterator over the entries.  
595 \*  
596 \* @return an iterator over the entries  
597 \*/  
598 public Iterator<Map.Entry<Object, Object>> entryIterator() {  
599 final Iterator<String> iter = keyIterator();  
600 return new Iterator<Map.Entry<Object, Object>>() {  
601 public boolean hasNext() {  
602 return iter.hasNext();  
603 }  
604 public Map.Entry<Object, Object> next() {  
605 final Object key = iter.next();  
606 final Object value = get(key);  
607 @SuppressWarnings("unchecked")  
608 final  
609 // This should not cause any problems; the key is actually a  
610 // string, but it does no harm to expose it as Object  
611 Map.Entry<Object, Object> tmpEntry = new Entry( BeanMap.this, key, value );  
612 return tmpEntry;  
613 }  
614 public void remove() {  
615 throw new UnsupportedOperationException( "remove() not supported for BeanMap" );  
616 }  
617 };  
618 }  
619  
620  
621 // Properties  
622 //-------------------------------------------------------------------------  
623  
624 /\*\*  
625 \* Returns the bean currently being operated on. The return value may  
626 \* be null if this map is empty.  
627 \*  
628 \* @return the bean being operated on by this map  
629 \*/  
630 public Object getBean() {  
631 return bean;  
632 }  
633  
634 /\*\*  
635 \* Sets the bean to be operated on by this map. The given value may  
636 \* be null, in which case this map will be empty.  
637 \*  
638 \* @param newBean the new bean to operate on  
639 \*/  
640 public void setBean( final Object newBean ) {  
641 bean = newBean;  
642 reinitialise();  
643 }  
644  
645 /\*\*  
646 \* Returns the accessor for the property with the given name.  
647 \*  
648 \* @param name the name of the property  
649 \* @return the accessor method for the property, or null  
650 \*/  
651 public Method getReadMethod(final String name) {  
652 return readMethods.get(name);  
653 }  
654  
655 /\*\*  
656 \* Returns the mutator for the property with the given name.  
657 \*  
658 \* @param name the name of the property  
659 \* @return the mutator method for the property, or null  
660 \*/  
661 public Method getWriteMethod(final String name) {  
662 return writeMethods.get(name);  
663 }  
664  
665  
666 // Implementation methods  
667 //-------------------------------------------------------------------------  
668  
669 /\*\*  
670 \* Returns the accessor for the property with the given name.  
671 \*  
672 \* @param name the name of the property  
673 \* @return null if the name is null; null if the name is not a  
674 \* {@link String}; null if no such property exists; or the accessor  
675 \* method for that property  
676 \*/  
677 protected Method getReadMethod( final Object name ) {  
678 return readMethods.get( name );  
679 }  
680  
681 /\*\*  
682 \* Returns the mutator for the property with the given name.  
683 \*  
684 \* @param name the name of the  
685 \* @return null if the name is null; null if the name is not a  
686 \* {@link String}; null if no such property exists; null if the  
687 \* property is read-only; or the mutator method for that property  
688 \*/  
689 protected Method getWriteMethod( final Object name ) {  
690 return writeMethods.get( name );  
691 }  
692  
693 /\*\*  
694 \* Reinitializes this bean. Called during {@link #setBean(Object)}.  
695 \* Does introspection to find properties.  
696 \*/  
697 protected void reinitialise() {  
698 readMethods.clear();  
699 writeMethods.clear();  
700 types.clear();  
701 initialise();  
702 }  
703  
704 private void initialise() {  
705 if(getBean() == null) {  
706 return;  
707 }  
708  
709 final Class<? extends Object> beanClass = getBean().getClass();  
710 try {  
711 //BeanInfo beanInfo = Introspector.getBeanInfo( bean, null );  
712 final BeanInfo beanInfo = Introspector.getBeanInfo( beanClass );  
713 final PropertyDescriptor[] propertyDescriptors = beanInfo.getPropertyDescriptors();  
714 if ( propertyDescriptors != null ) {  
715 for (final PropertyDescriptor propertyDescriptor : propertyDescriptors) {  
716 if ( propertyDescriptor != null ) {  
717 final String name = propertyDescriptor.getName();  
718 final Method readMethod = propertyDescriptor.getReadMethod();  
719 final Method writeMethod = propertyDescriptor.getWriteMethod();  
720 final Class<? extends Object> aType = propertyDescriptor.getPropertyType();  
721  
722 if ( readMethod != null ) {  
723 readMethods.put( name, readMethod );  
724 }  
725 if ( writeMethod != null ) {  
726 writeMethods.put( name, writeMethod );  
727 }  
728 types.put( name, aType );  
729 }  
730 }  
731 }  
732 }  
733 catch ( final IntrospectionException e ) {  
734 logWarn( e );  
735 }  
736 }  
737  
738 /\*\*  
739 \* Called during a successful {@link #put(Object,Object)} operation.  
740 \* Default implementation does nothing. Override to be notified of  
741 \* property changes in the bean caused by this map.  
742 \*  
743 \* @param key the name of the property that changed  
744 \* @param oldValue the old value for that property  
745 \* @param newValue the new value for that property  
746 \*/  
747 protected void firePropertyChange( final Object key, final Object oldValue, final Object newValue ) {  
748 }  
749  
750 // Implementation classes  
751 //-------------------------------------------------------------------------  
752  
753 /\*\*  
754 \* Map entry used by {@link BeanMap}.  
755 \*/  
756 protected static class Entry extends AbstractMapEntry {  
757 private final BeanMap owner;  
758  
759 /\*\*  
760 \* Constructs a new <code>Entry</code>.  
761 \*  
762 \* @param owner the BeanMap this entry belongs to  
763 \* @param key the key for this entry  
764 \* @param value the value for this entry  
765 \*/  
766 protected Entry( final BeanMap owner, final Object key, final Object value ) {  
767 super( key, value );  
768 this.owner = owner;  
769 }  
770  
771 /\*\*  
772 \* Sets the value.  
773 \*  
774 \* @param value the new value for the entry  
775 \* @return the old value for the entry  
776 \*/  
777 @Override  
778 public Object setValue(final Object value) {  
779 final Object key = getKey();  
780 final Object oldValue = owner.get( key );  
781  
782 owner.put( key, value );  
783 final Object newValue = owner.get( key );  
784 super.setValue( newValue );  
785 return oldValue;  
786 }  
787 }  
788  
789 /\*\*  
790 \* Creates an array of parameters to pass to the given mutator method.  
791 \* If the given object is not the right type to pass to the method  
792 \* directly, it will be converted using {@link #convertType(Class,Object)}.  
793 \*  
794 \* @param method the mutator method  
795 \* @param value the value to pass to the mutator method  
796 \* @return an array containing one object that is either the given value  
797 \* or a transformed value  
798 \* @throws IllegalAccessException if {@link #convertType(Class,Object)}  
799 \* raises it  
800 \* @throws IllegalArgumentException if any other exception is raised  
801 \* by {@link #convertType(Class,Object)}  
802 \* @throws ClassCastException if an error occurs creating the method args  
803 \*/  
804 protected Object[] createWriteMethodArguments( final Method method, Object value )  
805 throws IllegalAccessException, ClassCastException {  
806 try {  
807 if ( value != null ) {  
808 final Class<? extends Object>[] types = method.getParameterTypes();  
809 if ( types != null && types.length > 0 ) {  
810 final Class<? extends Object> paramType = types[0];  
811 if ( ! paramType.isAssignableFrom( value.getClass() ) ) {  
812 value = convertType( paramType, value );  
813 }  
814 }  
815 }  
816 final Object[] answer = { value };  
817 return answer;  
818 }  
819 catch ( final InvocationTargetException e ) {  
820 final IllegalArgumentException iae = new IllegalArgumentException(e.getMessage());  
821 if (BeanUtils.initCause(iae, e) == false) {  
822 logInfo(e);  
823 }  
824 throw iae;  
825 }  
826 catch ( final InstantiationException e ) {  
827 final IllegalArgumentException iae = new IllegalArgumentException(e.getMessage());  
828 if (BeanUtils.initCause(iae, e) == false) {  
829 logInfo(e);  
830 }  
831 BeanUtils.initCause(iae, e);  
832 throw iae;  
833 }  
834 }  
835  
836 /\*\*  
837 \* Converts the given value to the given type. First, reflection is  
838 \* is used to find a public constructor declared by the given class  
839 \* that takes one argument, which must be the precise type of the  
840 \* given value. If such a constructor is found, a new object is  
841 \* created by passing the given value to that constructor, and the  
842 \* newly constructed object is returned.<P>  
843 \*  
844 \* If no such constructor exists, and the given type is a primitive  
845 \* type, then the given value is converted to a string using its  
846 \* {@link Object#toString() toString()} method, and that string is  
847 \* parsed into the correct primitive type using, for instance,  
848 \* {@link Integer#valueOf(String)} to convert the string into an  
849 \* <code>int</code>.<P>  
850 \*  
851 \* If no special constructor exists and the given type is not a  
852 \* primitive type, this method returns the original value.  
853 \*  
854 \* @param newType the type to convert the value to  
855 \* @param value the value to convert  
856 \* @return the converted value  
857 \* @throws NumberFormatException if newType is a primitive type, and  
858 \* the string representation of the given value cannot be converted  
859 \* to that type  
860 \* @throws InstantiationException if the constructor found with  
861 \* reflection raises it  
862 \* @throws InvocationTargetException if the constructor found with  
863 \* reflection raises it  
864 \* @throws IllegalAccessException never  
865 \* @throws IllegalArgumentException never  
866 \*/  
867 protected Object convertType( final Class<?> newType, final Object value )  
868 throws InstantiationException, IllegalAccessException, IllegalArgumentException, InvocationTargetException {  
869  
870 // try call constructor  
871 final Class<?>[] types = { value.getClass() };  
872 try {  
873 final Constructor<?> constructor = newType.getConstructor( types );  
874 final Object[] arguments = { value };  
875 return constructor.newInstance( arguments );  
876 }  
877 catch ( final NoSuchMethodException e ) {  
878 // try using the transformers  
879 final Transformer transformer = getTypeTransformer( newType );  
880 if ( transformer != null ) {  
881 return transformer.transform( value );  
882 }  
883 return value;  
884 }  
885 }  
886  
887 /\*\*  
888 \* Returns a transformer for the given primitive type.  
889 \*  
890 \* @param aType the primitive type whose transformer to return  
891 \* @return a transformer that will convert strings into that type,  
892 \* or null if the given type is not a primitive type  
893 \*/  
894 protected Transformer getTypeTransformer( final Class<?> aType ) {  
895 return typeTransformers.get( aType );  
896 }  
897  
898 /\*\*  
899 \* Logs the given exception to <code>System.out</code>. Used to display  
900 \* warnings while accessing/mutating the bean.  
901 \*  
902 \* @param ex the exception to log  
903 \*/  
904 protected void logInfo(final Exception ex) {  
905 // Deliberately do not use LOG4J or Commons Logging to avoid dependencies  
906 System.out.println( "INFO: Exception: " + ex );  
907 }  
908  
909 /\*\*  
910 \* Logs the given exception to <code>System.err</code>. Used to display  
911 \* errors while accessing/mutating the bean.  
912 \*  
913 \* @param ex the exception to log  
914 \*/  
915 protected void logWarn(final Exception ex) {  
916 // Deliberately do not use LOG4J or Commons Logging to avoid dependencies  
917 System.out.println( "WARN: Exception: " + ex );  
918 ex.printStackTrace();  
919 }  
920}