Synchronized example

**package** com.corejava.practice;

**import** java.util.HashMap;

**import** java.util.Hashtable;

**public** **class** TestClass {

**public** **static** **void** main(String[] args)

{

A a = **new** B();

// B b = new B();

a.display();

//B.display();

}

}

**class** A {

**void** display(){

**synchronized** (**this**) {

System.*out*.println("A class..");

}

}

}

**class** B **extends** A {

**void** display(){

System.*out*.println("B class..");

}

}

**package** com.corejava.practice;

**import** java.util.HashMap;

**import** java.util.Hashtable;

**public** **class** TestClass {

**public** **static** **void** main(String[] args)

{

A a = **new** B();

// B b = new B();

a.display();

//B.display();

}

}

**class** A {

**void** display(){

System.*out*.println("A class..");

}

}

**class** B **extends** A {

**void** display(){

System.*out*.println("B class..");

}

}

Static method can’t be override

**package** com.corejava.practice;

**import** java.util.HashMap;

**import** java.util.Hashtable;

**public** **class** TestClass {

**public** **static** **void** main(String[] args)

{

A a = **new** B();

// B b = new B();

a.*display*();

//B.display();

}

}

**class** A {

**static** **void** display(){

System.*out*.println("A class..");

}

}

**class** B **extends** A {

**static** **void** display(){

System.*out*.println("B class..");

}

}

**package** com.corejava.practice;

**import** java.util.HashMap;

**import** java.util.Hashtable;

**public** **class** TestClass {

**public** **static** **void** main(String[] args)

{

A a = **new** B();

// B b = new B();

A.*display*();

//B.display();

}

}

**class** A {

**static** **void** display(){

System.*out*.println("A class..");

}

}

**class** B **extends** A {

**static** **void** display(){

System.*out*.println("B class..");

}

}

**package** com.corejava.practice;

**import** java.util.HashMap;

**import** java.util.Hashtable;

**public** **class** TestClass {

**public** **static** **void** main(String[] args)

{

A a = **new** B();

B b = **new** B();

A.*display*();

B.*display*();

}

}

**class** A {

**static** **void** display(){

System.*out*.println("A class..");

}

}

**class** B **extends** A {

**static** **void** display(){

System.*out*.println("B class..");

}

}

hashMap implementation

/\*

\* @(#)HashMap.java 1.73 07/03/13

\*

\* Copyright 2006 Sun Microsystems, Inc. All rights reserved.

\* SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.

\*/

package java.util;

import java.io.\*;

/\*\*

\* Hash table based implementation of the <tt>Map</tt> interface. This

\* implementation provides all of the optional map operations, and permits

\* <tt>null</tt> values and the <tt>null</tt> key. (The <tt>HashMap</tt>

\* class is roughly equivalent to <tt>Hashtable</tt>, except that it is

\* unsynchronized and permits nulls.) This class makes no guarantees as to

\* the order of the map; in particular, it does not guarantee that the order

\* will remain constant over time.

\*

\* <p>This implementation provides constant-time performance for the basic

\* operations (<tt>get</tt> and <tt>put</tt>), assuming the hash function

\* disperses the elements properly among the buckets. Iteration over

\* collection views requires time proportional to the "capacity" of the

\* <tt>HashMap</tt> instance (the number of buckets) plus its size (the number

\* of key-value mappings). Thus, it's very important not to set the initial

\* capacity too high (or the load factor too low) if iteration performance is

\* important.

\*

\* <p>An instance of <tt>HashMap</tt> has two parameters that affect its

\* performance: <i>initial capacity</i> and <i>load factor</i>. The

\* <i>capacity</i> is the number of buckets in the hash table, and the initial

\* capacity is simply the capacity at the time the hash table is created. The

\* <i>load factor</i> is a measure of how full the hash table is allowed to

\* get before its capacity is automatically increased. When the number of

\* entries in the hash table exceeds the product of the load factor and the

\* current capacity, the hash table is <i>rehashed</i> (that is, internal data

\* structures are rebuilt) so that the hash table has approximately twice the

\* number of buckets.

\*

\* <p>As a general rule, the default load factor (.75) offers a good tradeoff

\* between time and space costs. Higher values decrease the space overhead

\* but increase the lookup cost (reflected in most of the operations of the

\* <tt>HashMap</tt> class, including <tt>get</tt> and <tt>put</tt>). The

\* expected number of entries in the map and its load factor should be taken

\* into account when setting its initial capacity, so as to minimize the

\* number of rehash operations. If the initial capacity is greater

\* than the maximum number of entries divided by the load factor, no

\* rehash operations will ever occur.

\*

\* <p>If many mappings are to be stored in a <tt>HashMap</tt> instance,

\* creating it with a sufficiently large capacity will allow the mappings to

\* be stored more efficiently than letting it perform automatic rehashing as

\* needed to grow the table.

\*

\* <p><strong>Note that this implementation is not synchronized.</strong>

\* If multiple threads access a hash map concurrently, and at least one of

\* the threads modifies the map structurally, it <i>must</i> be

\* synchronized externally. (A structural modification is any operation

\* that adds or deletes one or more mappings; merely changing the value

\* associated with a key that an instance already contains is not a

\* structural modification.) This is typically accomplished by

\* synchronizing on some object that naturally encapsulates the map.

\*

\* If no such object exists, the map should be "wrapped" using the

\* {@link Collections#synchronizedMap Collections.synchronizedMap}

\* method. This is best done at creation time, to prevent accidental

\* unsynchronized access to the map:<pre>

\* Map m = Collections.synchronizedMap(new HashMap(...));</pre>

\*

\* <p>The iterators returned by all of this class's "collection view methods"

\* are <i>fail-fast</i>: if the map is structurally modified at any time after

\* the iterator is created, in any way except through the iterator's own

\* <tt>remove</tt> method, the iterator will throw a

\* {@link ConcurrentModificationException}. Thus, in the face of concurrent

\* modification, the iterator fails quickly and cleanly, rather than risking

\* arbitrary, non-deterministic behavior at an undetermined time in the

\* future.

\*

\* <p>Note that the fail-fast behavior of an iterator cannot be guaranteed

\* as it is, generally speaking, impossible to make any hard guarantees in the

\* presence of unsynchronized concurrent modification. Fail-fast iterators

\* throw <tt>ConcurrentModificationException</tt> on a best-effort basis.

\* Therefore, it would be wrong to write a program that depended on this

\* exception for its correctness: <i>the fail-fast behavior of iterators

\* should be used only to detect bugs.</i>

\*

\* <p>This class is a member of the

\* <a href="{@docRoot}/../technotes/guides/collections/index.html">

\* Java Collections Framework</a>.

\*

\* @param <K> the type of keys maintained by this map

\* @param <V> the type of mapped values

\*

\* @author Doug Lea

\* @author Josh Bloch

\* @author Arthur van Hoff

\* @author Neal Gafter

\* @version 1.73, 03/13/07

\* @see Object#hashCode()

\* @see Collection

\* @see Map

\* @see TreeMap

\* @see Hashtable

\* @since 1.2

\*/

public class HashMap<K,V>

extends AbstractMap<K,V>

implements Map<K,V>, Cloneable, Serializable

{

/\*\*

\* The default initial capacity - MUST be a power of two.

\*/

static final int DEFAULT\_INITIAL\_CAPACITY = 16;

/\*\*

\* The maximum capacity, used if a higher value is implicitly specified

\* by either of the constructors with arguments.

\* MUST be a power of two <= 1<<30.

\*/

static final int MAXIMUM\_CAPACITY = 1 << 30;

/\*\*

\* The load factor used when none specified in constructor.

\*/

static final float DEFAULT\_LOAD\_FACTOR = 0.75f;

/\*\*

\* The table, resized as necessary. Length MUST Always be a power of two.

\*/

transient Entry[] table;

/\*\*

\* The number of key-value mappings contained in this map.

\*/

transient int size;

/\*\*

\* The next size value at which to resize (capacity \* load factor).

\* @serial

\*/

int threshold;

/\*\*

\* The load factor for the hash table.

\*

\* @serial

\*/

final float loadFactor;

/\*\*

\* The number of times this HashMap has been structurally modified

\* Structural modifications are those that change the number of mappings in

\* the HashMap or otherwise modify its internal structure (e.g.,

\* rehash). This field is used to make iterators on Collection-views of

\* the HashMap fail-fast. (See ConcurrentModificationException).

\*/

transient volatile int modCount;

/\*\*

\* Constructs an empty <tt>HashMap</tt> with the specified initial

\* capacity and load factor.

\*

\* @param initialCapacity the initial capacity

\* @param loadFactor the load factor

\* @throws IllegalArgumentException if the initial capacity is negative

\* or the load factor is nonpositive

\*/

public HashMap(int initialCapacity, float loadFactor) {

if (initialCapacity < 0)

throw new IllegalArgumentException("Illegal initial capacity: " +

initialCapacity);

if (initialCapacity > MAXIMUM\_CAPACITY)

initialCapacity = MAXIMUM\_CAPACITY;

if (loadFactor <= 0 || Float.isNaN(loadFactor))

throw new IllegalArgumentException("Illegal load factor: " +

loadFactor);

// Find a power of 2 >= initialCapacity

int capacity = 1;

while (capacity < initialCapacity)

capacity <<= 1;

this.loadFactor = loadFactor;

threshold = (int)(capacity \* loadFactor);

table = new Entry[capacity];

init();

}

/\*\*

\* Constructs an empty <tt>HashMap</tt> with the specified initial

\* capacity and the default load factor (0.75).

\*

\* @param initialCapacity the initial capacity.

\* @throws IllegalArgumentException if the initial capacity is negative.

\*/

public HashMap(int initialCapacity) {

this(initialCapacity, DEFAULT\_LOAD\_FACTOR);

}

/\*\*

\* Constructs an empty <tt>HashMap</tt> with the default initial capacity

\* (16) and the default load factor (0.75).

\*/

public HashMap() {

this.loadFactor = DEFAULT\_LOAD\_FACTOR;

threshold = (int)(DEFAULT\_INITIAL\_CAPACITY \* DEFAULT\_LOAD\_FACTOR);

table = new Entry[DEFAULT\_INITIAL\_CAPACITY];

init();

}

/\*\*

\* Constructs a new <tt>HashMap</tt> with the same mappings as the

\* specified <tt>Map</tt>. The <tt>HashMap</tt> is created with

\* default load factor (0.75) and an initial capacity sufficient to

\* hold the mappings in the specified <tt>Map</tt>.

\*

\* @param m the map whose mappings are to be placed in this map

\* @throws NullPointerException if the specified map is null

\*/

public HashMap(Map<? extends K, ? extends V> m) {

this(Math.max((int) (m.size() / DEFAULT\_LOAD\_FACTOR) + 1,

DEFAULT\_INITIAL\_CAPACITY), DEFAULT\_LOAD\_FACTOR);

putAllForCreate(m);

}

// internal utilities

/\*\*

\* Initialization hook for subclasses. This method is called

\* in all constructors and pseudo-constructors (clone, readObject)

\* after HashMap has been initialized but before any entries have

\* been inserted. (In the absence of this method, readObject would

\* require explicit knowledge of subclasses.)

\*/

void init() {

}

/\*\*

\* Applies a supplemental hash function to a given hashCode, which

\* defends against poor quality hash functions. This is critical

\* because HashMap uses power-of-two length hash tables, that

\* otherwise encounter collisions for hashCodes that do not differ

\* in lower bits. Note: Null keys always map to hash 0, thus index 0.

\*/

static int hash(int h) {

// This function ensures that hashCodes that differ only by

// constant multiples at each bit position have a bounded

// number of collisions (approximately 8 at default load factor).

h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

/\*\*

\* Returns index for hash code h.

\*/

static int indexFor(int h, int length) {

return h & (length-1);

}

/\*\*

\* Returns the number of key-value mappings in this map.

\*

\* @return the number of key-value mappings in this map

\*/

public int size() {

return size;

}

/\*\*

\* Returns <tt>true</tt> if this map contains no key-value mappings.

\*

\* @return <tt>true</tt> if this map contains no key-value mappings

\*/

public boolean isEmpty() {

return size == 0;

}

/\*\*

\* Returns the value to which the specified key is mapped,

\* or {@code null} if this map contains no mapping for the key.

\*

\* <p>More formally, if this map contains a mapping from a key

\* {@code k} to a value {@code v} such that {@code (key==null ? k==null :

\* key.equals(k))}, then this method returns {@code v}; otherwise

\* it returns {@code null}. (There can be at most one such mapping.)

\*

\* <p>A return value of {@code null} does not <i>necessarily</i>

\* indicate that the map contains no mapping for the key; it's also

\* possible that the map explicitly maps the key to {@code null}.

\* The {@link #containsKey containsKey} operation may be used to

\* distinguish these two cases.

\*

\* @see #put(Object, Object)

\*/

public V get(Object key) {

if (key == null)

return getForNullKey();

int hash = hash(key.hashCode());

for (Entry<K,V> e = table[indexFor(hash, table.length)];

e != null;

e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

return e.value;

}

return null;

}

/\*\*

\* Offloaded version of get() to look up null keys. Null keys map

\* to index 0. This null case is split out into separate methods

\* for the sake of performance in the two most commonly used

\* operations (get and put), but incorporated with conditionals in

\* others.

\*/

private V getForNullKey() {

for (Entry<K,V> e = table[0]; e != null; e = e.next) {

if (e.key == null)

return e.value;

}

return null;

}

/\*\*

\* Returns <tt>true</tt> if this map contains a mapping for the

\* specified key.

\*

\* @param key The key whose presence in this map is to be tested

\* @return <tt>true</tt> if this map contains a mapping for the specified

\* key.

\*/

public boolean containsKey(Object key) {

return getEntry(key) != null;

}

/\*\*

\* Returns the entry associated with the specified key in the

\* HashMap. Returns null if the HashMap contains no mapping

\* for the key.

\*/

final Entry<K,V> getEntry(Object key) {

int hash = (key == null) ? 0 : hash(key.hashCode());

for (Entry<K,V> e = table[indexFor(hash, table.length)];

e != null;

e = e.next) {

Object k;

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k))))

return e;

}

return null;

}

/\*\*

\* Associates the specified value with the specified key in this map.

\* If the map previously contained a mapping for the key, the old

\* value is replaced.

\*

\* @param key key with which the specified value is to be associated

\* @param value value to be associated with the specified key

\* @return the previous value associated with <tt>key</tt>, or

\* <tt>null</tt> if there was no mapping for <tt>key</tt>.

\* (A <tt>null</tt> return can also indicate that the map

\* previously associated <tt>null</tt> with <tt>key</tt>.)

\*/

public V put(K key, V value) {

if (key == null)

return putForNullKey(value);

int hash = hash(key.hashCode());

int i = indexFor(hash, table.length);

for (Entry<K,V> e = table[i]; e != null; e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(hash, key, value, i);

return null;

}

/\*\*

\* Offloaded version of put for null keys

\*/

private V putForNullKey(V value) {

for (Entry<K,V> e = table[0]; e != null; e = e.next) {

if (e.key == null) {

V oldValue = e.value;

e.value = value;

e.recordAccess(this);

return oldValue;

}

}

modCount++;

addEntry(0, null, value, 0);

return null;

}

/\*\*

\* This method is used instead of put by constructors and

\* pseudoconstructors (clone, readObject). It does not resize the table,

\* check for comodification, etc. It calls createEntry rather than

\* addEntry.

\*/

private void putForCreate(K key, V value) {

int hash = (key == null) ? 0 : hash(key.hashCode());

int i = indexFor(hash, table.length);

/\*\*

\* Look for preexisting entry for key. This will never happen for

\* clone or deserialize. It will only happen for construction if the

\* input Map is a sorted map whose ordering is inconsistent w/ equals.

\*/

for (Entry<K,V> e = table[i]; e != null; e = e.next) {

Object k;

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k)))) {

e.value = value;

return;

}

}

createEntry(hash, key, value, i);

}

private void putAllForCreate(Map<? extends K, ? extends V> m) {

for (Iterator<? extends Map.Entry<? extends K, ? extends V>> i = m.entrySet().iterator(); i.hasNext(); ) {

Map.Entry<? extends K, ? extends V> e = i.next();

putForCreate(e.getKey(), e.getValue());

}

}

/\*\*

\* Rehashes the contents of this map into a new array with a

\* larger capacity. This method is called automatically when the

\* number of keys in this map reaches its threshold.

\*

\* If current capacity is MAXIMUM\_CAPACITY, this method does not

\* resize the map, but sets threshold to Integer.MAX\_VALUE.

\* This has the effect of preventing future calls.

\*

\* @param newCapacity the new capacity, MUST be a power of two;

\* must be greater than current capacity unless current

\* capacity is MAXIMUM\_CAPACITY (in which case value

\* is irrelevant).

\*/

void resize(int newCapacity) {

Entry[] oldTable = table;

int oldCapacity = oldTable.length;

if (oldCapacity == MAXIMUM\_CAPACITY) {

threshold = Integer.MAX\_VALUE;

return;

}

Entry[] newTable = new Entry[newCapacity];

transfer(newTable);

table = newTable;

threshold = (int)(newCapacity \* loadFactor);

}

/\*\*

\* Transfers all entries from current table to newTable.

\*/

void transfer(Entry[] newTable) {

Entry[] src = table;

int newCapacity = newTable.length;

for (int j = 0; j < src.length; j++) {

Entry<K,V> e = src[j];

if (e != null) {

src[j] = null;

do {

Entry<K,V> next = e.next;

int i = indexFor(e.hash, newCapacity);

e.next = newTable[i];

newTable[i] = e;

e = next;

} while (e != null);

}

}

}

/\*\*

\* Copies all of the mappings from the specified map to this map.

\* These mappings will replace any mappings that this map had for

\* any of the keys currently in the specified map.

\*

\* @param m mappings to be stored in this map

\* @throws NullPointerException if the specified map is null

\*/

public void putAll(Map<? extends K, ? extends V> m) {

int numKeysToBeAdded = m.size();

if (numKeysToBeAdded == 0)

return;

/\*

\* Expand the map if the map if the number of mappings to be added

\* is greater than or equal to threshold. This is conservative; the

\* obvious condition is (m.size() + size) >= threshold, but this

\* condition could result in a map with twice the appropriate capacity,

\* if the keys to be added overlap with the keys already in this map.

\* By using the conservative calculation, we subject ourself

\* to at most one extra resize.

\*/

if (numKeysToBeAdded > threshold) {

int targetCapacity = (int)(numKeysToBeAdded / loadFactor + 1);

if (targetCapacity > MAXIMUM\_CAPACITY)

targetCapacity = MAXIMUM\_CAPACITY;

int newCapacity = table.length;

while (newCapacity < targetCapacity)

newCapacity <<= 1;

if (newCapacity > table.length)

resize(newCapacity);

}

for (Iterator<? extends Map.Entry<? extends K, ? extends V>> i = m.entrySet().iterator(); i.hasNext(); ) {

Map.Entry<? extends K, ? extends V> e = i.next();

put(e.getKey(), e.getValue());

}

}

/\*\*

\* Removes the mapping for the specified key from this map if present.

\*

\* @param key key whose mapping is to be removed from the map

\* @return the previous value associated with <tt>key</tt>, or

\* <tt>null</tt> if there was no mapping for <tt>key</tt>.

\* (A <tt>null</tt> return can also indicate that the map

\* previously associated <tt>null</tt> with <tt>key</tt>.)

\*/

public V remove(Object key) {

Entry<K,V> e = removeEntryForKey(key);

return (e == null ? null : e.value);

}

/\*\*

\* Removes and returns the entry associated with the specified key

\* in the HashMap. Returns null if the HashMap contains no mapping

\* for this key.

\*/

final Entry<K,V> removeEntryForKey(Object key) {

int hash = (key == null) ? 0 : hash(key.hashCode());

int i = indexFor(hash, table.length);

Entry<K,V> prev = table[i];

Entry<K,V> e = prev;

while (e != null) {

Entry<K,V> next = e.next;

Object k;

if (e.hash == hash &&

((k = e.key) == key || (key != null && key.equals(k)))) {

modCount++;

size--;

if (prev == e)

table[i] = next;

else

prev.next = next;

e.recordRemoval(this);

return e;

}

prev = e;

e = next;

}

return e;

}

/\*\*

\* Special version of remove for EntrySet.

\*/

final Entry<K,V> removeMapping(Object o) {

if (!(o instanceof Map.Entry))

return null;

Map.Entry<K,V> entry = (Map.Entry<K,V>) o;

Object key = entry.getKey();

int hash = (key == null) ? 0 : hash(key.hashCode());

int i = indexFor(hash, table.length);

Entry<K,V> prev = table[i];

Entry<K,V> e = prev;

while (e != null) {

Entry<K,V> next = e.next;

if (e.hash == hash && e.equals(entry)) {

modCount++;

size--;

if (prev == e)

table[i] = next;

else

prev.next = next;

e.recordRemoval(this);

return e;

}

prev = e;

e = next;

}

return e;

}

/\*\*

\* Removes all of the mappings from this map.

\* The map will be empty after this call returns.

\*/

public void clear() {

modCount++;

Entry[] tab = table;

for (int i = 0; i < tab.length; i++)

tab[i] = null;

size = 0;

}

/\*\*

\* Returns <tt>true</tt> if this map maps one or more keys to the

\* specified value.

\*

\* @param value value whose presence in this map is to be tested

\* @return <tt>true</tt> if this map maps one or more keys to the

\* specified value

\*/

public boolean containsValue(Object value) {

if (value == null)

return containsNullValue();

Entry[] tab = table;

for (int i = 0; i < tab.length ; i++)

for (Entry e = tab[i] ; e != null ; e = e.next)

if (value.equals(e.value))

return true;

return false;

}

/\*\*

\* Special-case code for containsValue with null argument

\*/

private boolean containsNullValue() {

Entry[] tab = table;

for (int i = 0; i < tab.length ; i++)

for (Entry e = tab[i] ; e != null ; e = e.next)

if (e.value == null)

return true;

return false;

}

/\*\*

\* Returns a shallow copy of this <tt>HashMap</tt> instance: the keys and

\* values themselves are not cloned.

\*

\* @return a shallow copy of this map

\*/

public Object clone() {

HashMap<K,V> result = null;

try {

result = (HashMap<K,V>)super.clone();

} catch (CloneNotSupportedException e) {

// assert false;

}

result.table = new Entry[table.length];

result.entrySet = null;

result.modCount = 0;

result.size = 0;

result.init();

result.putAllForCreate(this);

return result;

}

static class Entry<K,V> implements Map.Entry<K,V> {

final K key;

V value;

Entry<K,V> next;

final int hash;

/\*\*

\* Creates new entry.

\*/

Entry(int h, K k, V v, Entry<K,V> n) {

value = v;

next = n;

key = k;

hash = h;

}

public final K getKey() {

return key;

}

public final V getValue() {

return value;

}

public final V setValue(V newValue) {

V oldValue = value;

value = newValue;

return oldValue;

}

public final boolean equals(Object o) {

if (!(o instanceof Map.Entry))

return false;

Map.Entry e = (Map.Entry)o;

Object k1 = getKey();

Object k2 = e.getKey();

if (k1 == k2 || (k1 != null && k1.equals(k2))) {

Object v1 = getValue();

Object v2 = e.getValue();

if (v1 == v2 || (v1 != null && v1.equals(v2)))

return true;

}

return false;

}

public final int hashCode() {

return (key==null ? 0 : key.hashCode()) ^

(value==null ? 0 : value.hashCode());

}

public final String toString() {

return getKey() + "=" + getValue();

}

/\*\*

\* This method is invoked whenever the value in an entry is

\* overwritten by an invocation of put(k,v) for a key k that's already

\* in the HashMap.

\*/

void recordAccess(HashMap<K,V> m) {

}

/\*\*

\* This method is invoked whenever the entry is

\* removed from the table.

\*/

void recordRemoval(HashMap<K,V> m) {

}

}

/\*\*

\* Adds a new entry with the specified key, value and hash code to

\* the specified bucket. It is the responsibility of this

\* method to resize the table if appropriate.

\*

\* Subclass overrides this to alter the behavior of put method.

\*/

void addEntry(int hash, K key, V value, int bucketIndex) {

Entry<K,V> e = table[bucketIndex];

table[bucketIndex] = new Entry<K,V>(hash, key, value, e);

if (size++ >= threshold)

resize(2 \* table.length);

}

/\*\*

\* Like addEntry except that this version is used when creating entries

\* as part of Map construction or "pseudo-construction" (cloning,

\* deserialization). This version needn't worry about resizing the table.

\*

\* Subclass overrides this to alter the behavior of HashMap(Map),

\* clone, and readObject.

\*/

void createEntry(int hash, K key, V value, int bucketIndex) {

Entry<K,V> e = table[bucketIndex];

table[bucketIndex] = new Entry<K,V>(hash, key, value, e);

size++;

}

private abstract class HashIterator<E> implements Iterator<E> {

Entry<K,V> next; // next entry to return

int expectedModCount; // For fast-fail

int index; // current slot

Entry<K,V> current; // current entry

HashIterator() {

expectedModCount = modCount;

if (size > 0) { // advance to first entry

Entry[] t = table;

while (index < t.length && (next = t[index++]) == null)

;

}

}

public final boolean hasNext() {

return next != null;

}

final Entry<K,V> nextEntry() {

if (modCount != expectedModCount)

throw new ConcurrentModificationException();

Entry<K,V> e = next;

if (e == null)

throw new NoSuchElementException();

if ((next = e.next) == null) {

Entry[] t = table;

while (index < t.length && (next = t[index++]) == null)

;

}

current = e;

return e;

}

public void remove() {

if (current == null)

throw new IllegalStateException();

if (modCount != expectedModCount)

throw new ConcurrentModificationException();

Object k = current.key;

current = null;

HashMap.this.removeEntryForKey(k);

expectedModCount = modCount;

}

}

private final class ValueIterator extends HashIterator<V> {

public V next() {

return nextEntry().value;

}

}

private final class KeyIterator extends HashIterator<K> {

public K next() {

return nextEntry().getKey();

}

}

private final class EntryIterator extends HashIterator<Map.Entry<K,V>> {

public Map.Entry<K,V> next() {

return nextEntry();

}

}

// Subclass overrides these to alter behavior of views' iterator() method

Iterator<K> newKeyIterator() {

return new KeyIterator();

}

Iterator<V> newValueIterator() {

return new ValueIterator();

}

Iterator<Map.Entry<K,V>> newEntryIterator() {

return new EntryIterator();

}

// Views

private transient Set<Map.Entry<K,V>> entrySet = null;

/\*\*

\* Returns a {@link Set} view of the keys contained in this map.

\* The set is backed by the map, so changes to the map are

\* reflected in the set, and vice-versa. If the map is modified

\* while an iteration over the set is in progress (except through

\* the iterator's own <tt>remove</tt> operation), the results of

\* the iteration are undefined. The set supports element removal,

\* which removes the corresponding mapping from the map, via the

\* <tt>Iterator.remove</tt>, <tt>Set.remove</tt>,

\* <tt>removeAll</tt>, <tt>retainAll</tt>, and <tt>clear</tt>

\* operations. It does not support the <tt>add</tt> or <tt>addAll</tt>

\* operations.

\*/

public Set<K> keySet() {

Set<K> ks = keySet;

return (ks != null ? ks : (keySet = new KeySet()));

}

private final class KeySet extends AbstractSet<K> {

public Iterator<K> iterator() {

return newKeyIterator();

}

public int size() {

return size;

}

public boolean contains(Object o) {

return containsKey(o);

}

public boolean remove(Object o) {

return HashMap.this.removeEntryForKey(o) != null;

}

public void clear() {

HashMap.this.clear();

}

}

/\*\*

\* Returns a {@link Collection} view of the values contained in this map.

\* The collection is backed by the map, so changes to the map are

\* reflected in the collection, and vice-versa. If the map is

\* modified while an iteration over the collection is in progress

\* (except through the iterator's own <tt>remove</tt> operation),

\* the results of the iteration are undefined. The collection

\* supports element removal, which removes the corresponding

\* mapping from the map, via the <tt>Iterator.remove</tt>,

\* <tt>Collection.remove</tt>, <tt>removeAll</tt>,

\* <tt>retainAll</tt> and <tt>clear</tt> operations. It does not

\* support the <tt>add</tt> or <tt>addAll</tt> operations.

\*/

public Collection<V> values() {

Collection<V> vs = values;

return (vs != null ? vs : (values = new Values()));

}

private final class Values extends AbstractCollection<V> {

public Iterator<V> iterator() {

return newValueIterator();

}

public int size() {

return size;

}

public boolean contains(Object o) {

return containsValue(o);

}

public void clear() {

HashMap.this.clear();

}

}

/\*\*

\* Returns a {@link Set} view of the mappings contained in this map.

\* The set is backed by the map, so changes to the map are

\* reflected in the set, and vice-versa. If the map is modified

\* while an iteration over the set is in progress (except through

\* the iterator's own <tt>remove</tt> operation, or through the

\* <tt>setValue</tt> operation on a map entry returned by the

\* iterator) the results of the iteration are undefined. The set

\* supports element removal, which removes the corresponding

\* mapping from the map, via the <tt>Iterator.remove</tt>,

\* <tt>Set.remove</tt>, <tt>removeAll</tt>, <tt>retainAll</tt> and

\* <tt>clear</tt> operations. It does not support the

\* <tt>add</tt> or <tt>addAll</tt> operations.

\*

\* @return a set view of the mappings contained in this map

\*/

public Set<Map.Entry<K,V>> entrySet() {

return entrySet0();

}

private Set<Map.Entry<K,V>> entrySet0() {

Set<Map.Entry<K,V>> es = entrySet;

return es != null ? es : (entrySet = new EntrySet());

}

private final class EntrySet extends AbstractSet<Map.Entry<K,V>> {

public Iterator<Map.Entry<K,V>> iterator() {

return newEntryIterator();

}

public boolean contains(Object o) {

if (!(o instanceof Map.Entry))

return false;

Map.Entry<K,V> e = (Map.Entry<K,V>) o;

Entry<K,V> candidate = getEntry(e.getKey());

return candidate != null && candidate.equals(e);

}

public boolean remove(Object o) {

return removeMapping(o) != null;

}

public int size() {

return size;

}

public void clear() {

HashMap.this.clear();

}

}

/\*\*

\* Save the state of the <tt>HashMap</tt> instance to a stream (i.e.,

\* serialize it).

\*

\* @serialData The <i>capacity</i> of the HashMap (the length of the

\* bucket array) is emitted (int), followed by the

\* <i>size</i> (an int, the number of key-value

\* mappings), followed by the key (Object) and value (Object)

\* for each key-value mapping. The key-value mappings are

\* emitted in no particular order.

\*/

private void writeObject(java.io.ObjectOutputStream s)

throws IOException

{

Iterator<Map.Entry<K,V>> i =

(size > 0) ? entrySet0().iterator() : null;

// Write out the threshold, loadfactor, and any hidden stuff

s.defaultWriteObject();

// Write out number of buckets

s.writeInt(table.length);

// Write out size (number of Mappings)

s.writeInt(size);

// Write out keys and values (alternating)

if (i != null) {

while (i.hasNext()) {

Map.Entry<K,V> e = i.next();

s.writeObject(e.getKey());

s.writeObject(e.getValue());

}

}

}

private static final long serialVersionUID = 362498820763181265L;

/\*\*

\* Reconstitute the <tt>HashMap</tt> instance from a stream (i.e.,

\* deserialize it).

\*/

private void readObject(java.io.ObjectInputStream s)

throws IOException, ClassNotFoundException

{

// Read in the threshold, loadfactor, and any hidden stuff

s.defaultReadObject();

// Read in number of buckets and allocate the bucket array;

int numBuckets = s.readInt();

table = new Entry[numBuckets];

init(); // Give subclass a chance to do its thing.

// Read in size (number of Mappings)

int size = s.readInt();

// Read the keys and values, and put the mappings in the HashMap

for (int i=0; i<size; i++) {

K key = (K) s.readObject();

V value = (V) s.readObject();

putForCreate(key, value);

}

}

// These methods are used when serializing HashSets

int capacity() { return table.length; }

float loadFactor() { return loadFactor; }

}

Hash table implementation

/\*

\* @(#)Hashtable.java 1.116 06/05/26

\*

\* Copyright 2006 Sun Microsystems, Inc. All rights reserved.

\* SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.

\*/

package java.util;

import java.io.\*;

/\*\*

\* This class implements a hashtable, which maps keys to values. Any

\* non-<code>null</code> object can be used as a key or as a value. <p>

\*

\* To successfully store and retrieve objects from a hashtable, the

\* objects used as keys must implement the <code>hashCode</code>

\* method and the <code>equals</code> method. <p>

\*

\* An instance of <code>Hashtable</code> has two parameters that affect its

\* performance: <i>initial capacity</i> and <i>load factor</i>. The

\* <i>capacity</i> is the number of <i>buckets</i> in the hash table, and the

\* <i>initial capacity</i> is simply the capacity at the time the hash table

\* is created. Note that the hash table is <i>open</i>: in the case of a "hash

\* collision", a single bucket stores multiple entries, which must be searched

\* sequentially. The <i>load factor</i> is a measure of how full the hash

\* table is allowed to get before its capacity is automatically increased.

\* The initial capacity and load factor parameters are merely hints to

\* the implementation. The exact details as to when and whether the rehash

\* method is invoked are implementation-dependent.<p>

\*

\* Generally, the default load factor (.75) offers a good tradeoff between

\* time and space costs. Higher values decrease the space overhead but

\* increase the time cost to look up an entry (which is reflected in most

\* <tt>Hashtable</tt> operations, including <tt>get</tt> and <tt>put</tt>).<p>

\*

\* The initial capacity controls a tradeoff between wasted space and the

\* need for <code>rehash</code> operations, which are time-consuming.

\* No <code>rehash</code> operations will <i>ever</i> occur if the initial

\* capacity is greater than the maximum number of entries the

\* <tt>Hashtable</tt> will contain divided by its load factor. However,

\* setting the initial capacity too high can waste space.<p>

\*

\* If many entries are to be made into a <code>Hashtable</code>,

\* creating it with a sufficiently large capacity may allow the

\* entries to be inserted more efficiently than letting it perform

\* automatic rehashing as needed to grow the table. <p>

\*

\* This example creates a hashtable of numbers. It uses the names of

\* the numbers as keys:

\* <pre> {@code

\* Hashtable<String, Integer> numbers

\* = new Hashtable<String, Integer>();

\* numbers.put("one", 1);

\* numbers.put("two", 2);

\* numbers.put("three", 3);}</pre>

\*

\* <p>To retrieve a number, use the following code:

\* <pre> {@code

\* Integer n = numbers.get("two");

\* if (n != null) {

\* System.out.println("two = " + n);

\* }}</pre>

\*

\* <p>The iterators returned by the <tt>iterator</tt> method of the collections

\* returned by all of this class's "collection view methods" are

\* <em>fail-fast</em>: if the Hashtable is structurally modified at any time

\* after the iterator is created, in any way except through the iterator's own

\* <tt>remove</tt> method, the iterator will throw a {@link

\* ConcurrentModificationException}. Thus, in the face of concurrent

\* modification, the iterator fails quickly and cleanly, rather than risking

\* arbitrary, non-deterministic behavior at an undetermined time in the future.

\* The Enumerations returned by Hashtable's keys and elements methods are

\* <em>not</em> fail-fast.

\*

\* <p>Note that the fail-fast behavior of an iterator cannot be guaranteed

\* as it is, generally speaking, impossible to make any hard guarantees in the

\* presence of unsynchronized concurrent modification. Fail-fast iterators

\* throw <tt>ConcurrentModificationException</tt> on a best-effort basis.

\* Therefore, it would be wrong to write a program that depended on this

\* exception for its correctness: <i>the fail-fast behavior of iterators

\* should be used only to detect bugs.</i>

\*

\* <p>As of the Java 2 platform v1.2, this class was retrofitted to

\* implement the {@link Map} interface, making it a member of the

\* <a href="{@docRoot}/../technotes/guides/collections/index.html"> Java

\* Collections Framework</a>. Unlike the new collection

\* implementations, {@code Hashtable} is synchronized.

\*

\* @author Arthur van Hoff

\* @author Josh Bloch

\* @author Neal Gafter

\* @version 1.116, 05/26/06

\* @see Object#equals(java.lang.Object)

\* @see Object#hashCode()

\* @see Hashtable#rehash()

\* @see Collection

\* @see Map

\* @see HashMap

\* @see TreeMap

\* @since JDK1.0

\*/

public class Hashtable<K,V>

extends Dictionary<K,V>

implements Map<K,V>, Cloneable, java.io.Serializable {

/\*\*

\* The hash table data.

\*/

private transient Entry[] table;

/\*\*

\* The total number of entries in the hash table.

\*/

private transient int count;

/\*\*

\* The table is rehashed when its size exceeds this threshold. (The

\* value of this field is (int)(capacity \* loadFactor).)

\*

\* @serial

\*/

private int threshold;

/\*\*

\* The load factor for the hashtable.

\*

\* @serial

\*/

private float loadFactor;

/\*\*

\* The number of times this Hashtable has been structurally modified

\* Structural modifications are those that change the number of entries in

\* the Hashtable or otherwise modify its internal structure (e.g.,

\* rehash). This field is used to make iterators on Collection-views of

\* the Hashtable fail-fast. (See ConcurrentModificationException).

\*/

private transient int modCount = 0;

/\*\* use serialVersionUID from JDK 1.0.2 for interoperability \*/

private static final long serialVersionUID = 1421746759512286392L;

/\*\*

\* Constructs a new, empty hashtable with the specified initial

\* capacity and the specified load factor.

\*

\* @param initialCapacity the initial capacity of the hashtable.

\* @param loadFactor the load factor of the hashtable.

\* @exception IllegalArgumentException if the initial capacity is less

\* than zero, or if the load factor is nonpositive.

\*/

public Hashtable(int initialCapacity, float loadFactor) {

if (initialCapacity < 0)

throw new IllegalArgumentException("Illegal Capacity: "+

initialCapacity);

if (loadFactor <= 0 || Float.isNaN(loadFactor))

throw new IllegalArgumentException("Illegal Load: "+loadFactor);

if (initialCapacity==0)

initialCapacity = 1;

this.loadFactor = loadFactor;

table = new Entry[initialCapacity];

threshold = (int)(initialCapacity \* loadFactor);

}

/\*\*

\* Constructs a new, empty hashtable with the specified initial capacity

\* and default load factor (0.75).

\*

\* @param initialCapacity the initial capacity of the hashtable.

\* @exception IllegalArgumentException if the initial capacity is less

\* than zero.

\*/

public Hashtable(int initialCapacity) {

this(initialCapacity, 0.75f);

}

/\*\*

\* Constructs a new, empty hashtable with a default initial capacity (11)

\* and load factor (0.75).

\*/

public Hashtable() {

this(11, 0.75f);

}

/\*\*

\* Constructs a new hashtable with the same mappings as the given

\* Map. The hashtable is created with an initial capacity sufficient to

\* hold the mappings in the given Map and a default load factor (0.75).

\*

\* @param t the map whose mappings are to be placed in this map.

\* @throws NullPointerException if the specified map is null.

\* @since 1.2

\*/

public Hashtable(Map<? extends K, ? extends V> t) {

this(Math.max(2\*t.size(), 11), 0.75f);

putAll(t);

}

/\*\*

\* Returns the number of keys in this hashtable.

\*

\* @return the number of keys in this hashtable.

\*/

public synchronized int size() {

return count;

}

/\*\*

\* Tests if this hashtable maps no keys to values.

\*

\* @return <code>true</code> if this hashtable maps no keys to values;

\* <code>false</code> otherwise.

\*/

public synchronized boolean isEmpty() {

return count == 0;

}

/\*\*

\* Returns an enumeration of the keys in this hashtable.

\*

\* @return an enumeration of the keys in this hashtable.

\* @see Enumeration

\* @see #elements()

\* @see #keySet()

\* @see Map

\*/

public synchronized Enumeration<K> keys() {

return this.<K>getEnumeration(KEYS);

}

/\*\*

\* Returns an enumeration of the values in this hashtable.

\* Use the Enumeration methods on the returned object to fetch the elements

\* sequentially.

\*

\* @return an enumeration of the values in this hashtable.

\* @see java.util.Enumeration

\* @see #keys()

\* @see #values()

\* @see Map

\*/

public synchronized Enumeration<V> elements() {

return this.<V>getEnumeration(VALUES);

}

/\*\*

\* Tests if some key maps into the specified value in this hashtable.

\* This operation is more expensive than the {@link #containsKey

\* containsKey} method.

\*

\* <p>Note that this method is identical in functionality to

\* {@link #containsValue containsValue}, (which is part of the

\* {@link Map} interface in the collections framework).

\*

\* @param value a value to search for

\* @return <code>true</code> if and only if some key maps to the

\* <code>value</code> argument in this hashtable as

\* determined by the <tt>equals</tt> method;

\* <code>false</code> otherwise.

\* @exception NullPointerException if the value is <code>null</code>

\*/

public synchronized boolean contains(Object value) {

if (value == null) {

throw new NullPointerException();

}

Entry tab[] = table;

for (int i = tab.length ; i-- > 0 ;) {

for (Entry<K,V> e = tab[i] ; e != null ; e = e.next) {

if (e.value.equals(value)) {

return true;

}

}

}

return false;

}

/\*\*

\* Returns true if this hashtable maps one or more keys to this value.

\*

\* <p>Note that this method is identical in functionality to {@link

\* #contains contains} (which predates the {@link Map} interface).

\*

\* @param value value whose presence in this hashtable is to be tested

\* @return <tt>true</tt> if this map maps one or more keys to the

\* specified value

\* @throws NullPointerException if the value is <code>null</code>

\* @since 1.2

\*/

public boolean containsValue(Object value) {

return contains(value);

}

/\*\*

\* Tests if the specified object is a key in this hashtable.

\*

\* @param key possible key

\* @return <code>true</code> if and only if the specified object

\* is a key in this hashtable, as determined by the

\* <tt>equals</tt> method; <code>false</code> otherwise.

\* @throws NullPointerException if the key is <code>null</code>

\* @see #contains(Object)

\*/

public synchronized boolean containsKey(Object key) {

Entry tab[] = table;

int hash = key.hashCode();

int index = (hash & 0x7FFFFFFF) % tab.length;

for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {

if ((e.hash == hash) && e.key.equals(key)) {

return true;

}

}

return false;

}

/\*\*

\* Returns the value to which the specified key is mapped,

\* or {@code null} if this map contains no mapping for the key.

\*

\* <p>More formally, if this map contains a mapping from a key

\* {@code k} to a value {@code v} such that {@code (key.equals(k))},

\* then this method returns {@code v}; otherwise it returns

\* {@code null}. (There can be at most one such mapping.)

\*

\* @param key the key whose associated value is to be returned

\* @return the value to which the specified key is mapped, or

\* {@code null} if this map contains no mapping for the key

\* @throws NullPointerException if the specified key is null

\* @see #put(Object, Object)

\*/

public synchronized V get(Object key) {

Entry tab[] = table;

int hash = key.hashCode();

int index = (hash & 0x7FFFFFFF) % tab.length;

for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {

if ((e.hash == hash) && e.key.equals(key)) {

return e.value;

}

}

return null;

}

/\*\*

\* Increases the capacity of and internally reorganizes this

\* hashtable, in order to accommodate and access its entries more

\* efficiently. This method is called automatically when the

\* number of keys in the hashtable exceeds this hashtable's capacity

\* and load factor.

\*/

protected void rehash() {

int oldCapacity = table.length;

Entry[] oldMap = table;

int newCapacity = oldCapacity \* 2 + 1;

Entry[] newMap = new Entry[newCapacity];

modCount++;

threshold = (int)(newCapacity \* loadFactor);

table = newMap;

for (int i = oldCapacity ; i-- > 0 ;) {

for (Entry<K,V> old = oldMap[i] ; old != null ; ) {

Entry<K,V> e = old;

old = old.next;

int index = (e.hash & 0x7FFFFFFF) % newCapacity;

e.next = newMap[index];

newMap[index] = e;

}

}

}

/\*\*

\* Maps the specified <code>key</code> to the specified

\* <code>value</code> in this hashtable. Neither the key nor the

\* value can be <code>null</code>. <p>

\*

\* The value can be retrieved by calling the <code>get</code> method

\* with a key that is equal to the original key.

\*

\* @param key the hashtable key

\* @param value the value

\* @return the previous value of the specified key in this hashtable,

\* or <code>null</code> if it did not have one

\* @exception NullPointerException if the key or value is

\* <code>null</code>

\* @see Object#equals(Object)

\* @see #get(Object)

\*/

public synchronized V put(K key, V value) {

// Make sure the value is not null

if (value == null) {

throw new NullPointerException();

}

// Makes sure the key is not already in the hashtable.

Entry tab[] = table;

int hash = key.hashCode();

int index = (hash & 0x7FFFFFFF) % tab.length;

for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {

if ((e.hash == hash) && e.key.equals(key)) {

V old = e.value;

e.value = value;

return old;

}

}

modCount++;

if (count >= threshold) {

// Rehash the table if the threshold is exceeded

rehash();

tab = table;

index = (hash & 0x7FFFFFFF) % tab.length;

}

// Creates the new entry.

Entry<K,V> e = tab[index];

tab[index] = new Entry<K,V>(hash, key, value, e);

count++;

return null;

}

/\*\*

\* Removes the key (and its corresponding value) from this

\* hashtable. This method does nothing if the key is not in the hashtable.

\*

\* @param key the key that needs to be removed

\* @return the value to which the key had been mapped in this hashtable,

\* or <code>null</code> if the key did not have a mapping

\* @throws NullPointerException if the key is <code>null</code>

\*/

public synchronized V remove(Object key) {

Entry tab[] = table;

int hash = key.hashCode();

int index = (hash & 0x7FFFFFFF) % tab.length;

for (Entry<K,V> e = tab[index], prev = null ; e != null ; prev = e, e = e.next) {

if ((e.hash == hash) && e.key.equals(key)) {

modCount++;

if (prev != null) {

prev.next = e.next;

} else {

tab[index] = e.next;

}

count--;

V oldValue = e.value;

e.value = null;

return oldValue;

}

}

return null;

}

/\*\*

\* Copies all of the mappings from the specified map to this hashtable.

\* These mappings will replace any mappings that this hashtable had for any

\* of the keys currently in the specified map.

\*

\* @param t mappings to be stored in this map

\* @throws NullPointerException if the specified map is null

\* @since 1.2

\*/

public synchronized void putAll(Map<? extends K, ? extends V> t) {

for (Map.Entry<? extends K, ? extends V> e : t.entrySet())

put(e.getKey(), e.getValue());

}

/\*\*

\* Clears this hashtable so that it contains no keys.

\*/

public synchronized void clear() {

Entry tab[] = table;

modCount++;

for (int index = tab.length; --index >= 0; )

tab[index] = null;

count = 0;

}

/\*\*

\* Creates a shallow copy of this hashtable. All the structure of the

\* hashtable itself is copied, but the keys and values are not cloned.

\* This is a relatively expensive operation.

\*

\* @return a clone of the hashtable

\*/

public synchronized Object clone() {

try {

Hashtable<K,V> t = (Hashtable<K,V>) super.clone();

t.table = new Entry[table.length];

for (int i = table.length ; i-- > 0 ; ) {

t.table[i] = (table[i] != null)

? (Entry<K,V>) table[i].clone() : null;

}

t.keySet = null;

t.entrySet = null;

t.values = null;

t.modCount = 0;

return t;

} catch (CloneNotSupportedException e) {

// this shouldn't happen, since we are Cloneable

throw new InternalError();

}

}

/\*\*

\* Returns a string representation of this <tt>Hashtable</tt> object

\* in the form of a set of entries, enclosed in braces and separated

\* by the ASCII characters "<tt>,&nbsp;</tt>" (comma and space). Each

\* entry is rendered as the key, an equals sign <tt>=</tt>, and the

\* associated element, where the <tt>toString</tt> method is used to

\* convert the key and element to strings.

\*

\* @return a string representation of this hashtable

\*/

public synchronized String toString() {

int max = size() - 1;

if (max == -1)

return "{}";

StringBuilder sb = new StringBuilder();

Iterator<Map.Entry<K,V>> it = entrySet().iterator();

sb.append('{');

for (int i = 0; ; i++) {

Map.Entry<K,V> e = it.next();

K key = e.getKey();

V value = e.getValue();

sb.append(key == this ? "(this Map)" : key.toString());

sb.append('=');

sb.append(value == this ? "(this Map)" : value.toString());

if (i == max)

return sb.append('}').toString();

sb.append(", ");

}

}

private <T> Enumeration<T> getEnumeration(int type) {

if (count == 0) {

return (Enumeration<T>)emptyEnumerator;

} else {

return new Enumerator<T>(type, false);

}

}

private <T> Iterator<T> getIterator(int type) {

if (count == 0) {

return (Iterator<T>) emptyIterator;

} else {

return new Enumerator<T>(type, true);

}

}

// Views

/\*\*

\* Each of these fields are initialized to contain an instance of the

\* appropriate view the first time this view is requested. The views are

\* stateless, so there's no reason to create more than one of each.

\*/

private transient volatile Set<K> keySet = null;

private transient volatile Set<Map.Entry<K,V>> entrySet = null;

private transient volatile Collection<V> values = null;

/\*\*

\* Returns a {@link Set} view of the keys contained in this map.

\* The set is backed by the map, so changes to the map are

\* reflected in the set, and vice-versa. If the map is modified

\* while an iteration over the set is in progress (except through

\* the iterator's own <tt>remove</tt> operation), the results of

\* the iteration are undefined. The set supports element removal,

\* which removes the corresponding mapping from the map, via the

\* <tt>Iterator.remove</tt>, <tt>Set.remove</tt>,

\* <tt>removeAll</tt>, <tt>retainAll</tt>, and <tt>clear</tt>

\* operations. It does not support the <tt>add</tt> or <tt>addAll</tt>

\* operations.

\*

\* @since 1.2

\*/

public Set<K> keySet() {

if (keySet == null)

keySet = Collections.synchronizedSet(new KeySet(), this);

return keySet;

}

private class KeySet extends AbstractSet<K> {

public Iterator<K> iterator() {

return getIterator(KEYS);

}

public int size() {

return count;

}

public boolean contains(Object o) {

return containsKey(o);

}

public boolean remove(Object o) {

return Hashtable.this.remove(o) != null;

}

public void clear() {

Hashtable.this.clear();

}

}

/\*\*

\* Returns a {@link Set} view of the mappings contained in this map.

\* The set is backed by the map, so changes to the map are

\* reflected in the set, and vice-versa. If the map is modified

\* while an iteration over the set is in progress (except through

\* the iterator's own <tt>remove</tt> operation, or through the

\* <tt>setValue</tt> operation on a map entry returned by the

\* iterator) the results of the iteration are undefined. The set

\* supports element removal, which removes the corresponding

\* mapping from the map, via the <tt>Iterator.remove</tt>,

\* <tt>Set.remove</tt>, <tt>removeAll</tt>, <tt>retainAll</tt> and

\* <tt>clear</tt> operations. It does not support the

\* <tt>add</tt> or <tt>addAll</tt> operations.

\*

\* @since 1.2

\*/

public Set<Map.Entry<K,V>> entrySet() {

if (entrySet==null)

entrySet = Collections.synchronizedSet(new EntrySet(), this);

return entrySet;

}

private class EntrySet extends AbstractSet<Map.Entry<K,V>> {

public Iterator<Map.Entry<K,V>> iterator() {

return getIterator(ENTRIES);

}

public boolean add(Map.Entry<K,V> o) {

return super.add(o);

}

public boolean contains(Object o) {

if (!(o instanceof Map.Entry))

return false;

Map.Entry entry = (Map.Entry)o;

Object key = entry.getKey();

Entry[] tab = table;

int hash = key.hashCode();

int index = (hash & 0x7FFFFFFF) % tab.length;

for (Entry e = tab[index]; e != null; e = e.next)

if (e.hash==hash && e.equals(entry))

return true;

return false;

}

public boolean remove(Object o) {

if (!(o instanceof Map.Entry))

return false;

Map.Entry<K,V> entry = (Map.Entry<K,V>) o;

K key = entry.getKey();

Entry[] tab = table;

int hash = key.hashCode();

int index = (hash & 0x7FFFFFFF) % tab.length;

for (Entry<K,V> e = tab[index], prev = null; e != null;

prev = e, e = e.next) {

if (e.hash==hash && e.equals(entry)) {

modCount++;

if (prev != null)

prev.next = e.next;

else

tab[index] = e.next;

count--;

e.value = null;

return true;

}

}

return false;

}

public int size() {

return count;

}

public void clear() {

Hashtable.this.clear();

}

}

/\*\*

\* Returns a {@link Collection} view of the values contained in this map.

\* The collection is backed by the map, so changes to the map are

\* reflected in the collection, and vice-versa. If the map is

\* modified while an iteration over the collection is in progress

\* (except through the iterator's own <tt>remove</tt> operation),

\* the results of the iteration are undefined. The collection

\* supports element removal, which removes the corresponding

\* mapping from the map, via the <tt>Iterator.remove</tt>,

\* <tt>Collection.remove</tt>, <tt>removeAll</tt>,

\* <tt>retainAll</tt> and <tt>clear</tt> operations. It does not

\* support the <tt>add</tt> or <tt>addAll</tt> operations.

\*

\* @since 1.2

\*/

public Collection<V> values() {

if (values==null)

values = Collections.synchronizedCollection(new ValueCollection(),

this);

return values;

}

private class ValueCollection extends AbstractCollection<V> {

public Iterator<V> iterator() {

return getIterator(VALUES);

}

public int size() {

return count;

}

public boolean contains(Object o) {

return containsValue(o);

}

public void clear() {

Hashtable.this.clear();

}

}

// Comparison and hashing

/\*\*

\* Compares the specified Object with this Map for equality,

\* as per the definition in the Map interface.

\*

\* @param o object to be compared for equality with this hashtable

\* @return true if the specified Object is equal to this Map

\* @see Map#equals(Object)

\* @since 1.2

\*/

public synchronized boolean equals(Object o) {

if (o == this)

return true;

if (!(o instanceof Map))

return false;

Map<K,V> t = (Map<K,V>) o;

if (t.size() != size())

return false;

try {

Iterator<Map.Entry<K,V>> i = entrySet().iterator();

while (i.hasNext()) {

Map.Entry<K,V> e = i.next();

K key = e.getKey();

V value = e.getValue();

if (value == null) {

if (!(t.get(key)==null && t.containsKey(key)))

return false;

} else {

if (!value.equals(t.get(key)))

return false;

}

}

} catch (ClassCastException unused) {

return false;

} catch (NullPointerException unused) {

return false;

}

return true;

}

/\*\*

\* Returns the hash code value for this Map as per the definition in the

\* Map interface.

\*

\* @see Map#hashCode()

\* @since 1.2

\*/

public synchronized int hashCode() {

/\*

\* This code detects the recursion caused by computing the hash code

\* of a self-referential hash table and prevents the stack overflow

\* that would otherwise result. This allows certain 1.1-era

\* applets with self-referential hash tables to work. This code

\* abuses the loadFactor field to do double-duty as a hashCode

\* in progress flag, so as not to worsen the space performance.

\* A negative load factor indicates that hash code computation is

\* in progress.

\*/

int h = 0;

if (count == 0 || loadFactor < 0)

return h; // Returns zero

loadFactor = -loadFactor; // Mark hashCode computation in progress

Entry[] tab = table;

for (int i = 0; i < tab.length; i++)

for (Entry e = tab[i]; e != null; e = e.next)

h += e.key.hashCode() ^ e.value.hashCode();

loadFactor = -loadFactor; // Mark hashCode computation complete

return h;

}

/\*\*

\* Save the state of the Hashtable to a stream (i.e., serialize it).

\*

\* @serialData The <i>capacity</i> of the Hashtable (the length of the

\* bucket array) is emitted (int), followed by the

\* <i>size</i> of the Hashtable (the number of key-value

\* mappings), followed by the key (Object) and value (Object)

\* for each key-value mapping represented by the Hashtable

\* The key-value mappings are emitted in no particular order.

\*/

private synchronized void writeObject(java.io.ObjectOutputStream s)

throws IOException

{

// Write out the length, threshold, loadfactor

s.defaultWriteObject();

// Write out length, count of elements and then the key/value objects

s.writeInt(table.length);

s.writeInt(count);

for (int index = table.length-1; index >= 0; index--) {

Entry entry = table[index];

while (entry != null) {

s.writeObject(entry.key);

s.writeObject(entry.value);

entry = entry.next;

}

}

}

/\*\*

\* Reconstitute the Hashtable from a stream (i.e., deserialize it).

\*/

private void readObject(java.io.ObjectInputStream s)

throws IOException, ClassNotFoundException

{

// Read in the length, threshold, and loadfactor

s.defaultReadObject();

// Read the original length of the array and number of elements

int origlength = s.readInt();

int elements = s.readInt();

// Compute new size with a bit of room 5% to grow but

// no larger than the original size. Make the length

// odd if it's large enough, this helps distribute the entries.

// Guard against the length ending up zero, that's not valid.

int length = (int)(elements \* loadFactor) + (elements / 20) + 3;

if (length > elements && (length & 1) == 0)

length--;

if (origlength > 0 && length > origlength)

length = origlength;

Entry[] table = new Entry[length];

count = 0;

// Read the number of elements and then all the key/value objects

for (; elements > 0; elements--) {

K key = (K)s.readObject();

V value = (V)s.readObject();

// synch could be eliminated for performance

reconstitutionPut(table, key, value);

}

this.table = table;

}

/\*\*

\* The put method used by readObject. This is provided because put

\* is overridable and should not be called in readObject since the

\* subclass will not yet be initialized.

\*

\* <p>This differs from the regular put method in several ways. No

\* checking for rehashing is necessary since the number of elements

\* initially in the table is known. The modCount is not incremented

\* because we are creating a new instance. Also, no return value

\* is needed.

\*/

private void reconstitutionPut(Entry[] tab, K key, V value)

throws StreamCorruptedException

{

if (value == null) {

throw new java.io.StreamCorruptedException();

}

// Makes sure the key is not already in the hashtable.

// This should not happen in deserialized version.

int hash = key.hashCode();

int index = (hash & 0x7FFFFFFF) % tab.length;

for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {

if ((e.hash == hash) && e.key.equals(key)) {

throw new java.io.StreamCorruptedException();

}

}

// Creates the new entry.

Entry<K,V> e = tab[index];

tab[index] = new Entry<K,V>(hash, key, value, e);

count++;

}

/\*\*

\* Hashtable collision list.

\*/

private static class Entry<K,V> implements Map.Entry<K,V> {

int hash;

K key;

V value;

Entry<K,V> next;

protected Entry(int hash, K key, V value, Entry<K,V> next) {

this.hash = hash;

this.key = key;

this.value = value;

this.next = next;

}

protected Object clone() {

return new Entry<K,V>(hash, key, value,

(next==null ? null : (Entry<K,V>) next.clone()));

}

// Map.Entry Ops

public K getKey() {

return key;

}

public V getValue() {

return value;

}

public V setValue(V value) {

if (value == null)

throw new NullPointerException();

V oldValue = this.value;

this.value = value;

return oldValue;

}

public boolean equals(Object o) {

if (!(o instanceof Map.Entry))

return false;

Map.Entry e = (Map.Entry)o;

return (key==null ? e.getKey()==null : key.equals(e.getKey())) &&

(value==null ? e.getValue()==null : value.equals(e.getValue()));

}

public int hashCode() {

return hash ^ (value==null ? 0 : value.hashCode());

}

public String toString() {

return key.toString()+"="+value.toString();

}

}

// Types of Enumerations/Iterations

private static final int KEYS = 0;

private static final int VALUES = 1;

private static final int ENTRIES = 2;

/\*\*

\* A hashtable enumerator class. This class implements both the

\* Enumeration and Iterator interfaces, but individual instances

\* can be created with the Iterator methods disabled. This is necessary

\* to avoid unintentionally increasing the capabilities granted a user

\* by passing an Enumeration.

\*/

private class Enumerator<T> implements Enumeration<T>, Iterator<T> {

Entry[] table = Hashtable.this.table;

int index = table.length;

Entry<K,V> entry = null;

Entry<K,V> lastReturned = null;

int type;

/\*\*

\* Indicates whether this Enumerator is serving as an Iterator

\* or an Enumeration. (true -> Iterator).

\*/

boolean iterator;

/\*\*

\* The modCount value that the iterator believes that the backing

\* Hashtable should have. If this expectation is violated, the iterator

\* has detected concurrent modification.

\*/

protected int expectedModCount = modCount;

Enumerator(int type, boolean iterator) {

this.type = type;

this.iterator = iterator;

}

public boolean hasMoreElements() {

Entry<K,V> e = entry;

int i = index;

Entry[] t = table;

/\* Use locals for faster loop iteration \*/

while (e == null && i > 0) {

e = t[--i];

}

entry = e;

index = i;

return e != null;

}

public T nextElement() {

Entry<K,V> et = entry;

int i = index;

Entry[] t = table;

/\* Use locals for faster loop iteration \*/

while (et == null && i > 0) {

et = t[--i];

}

entry = et;

index = i;

if (et != null) {

Entry<K,V> e = lastReturned = entry;

entry = e.next;

return type == KEYS ? (T)e.key : (type == VALUES ? (T)e.value : (T)e);

}

throw new NoSuchElementException("Hashtable Enumerator");

}

// Iterator methods

public boolean hasNext() {

return hasMoreElements();

}

public T next() {

if (modCount != expectedModCount)

throw new ConcurrentModificationException();

return nextElement();

}

public void remove() {

if (!iterator)

throw new UnsupportedOperationException();

if (lastReturned == null)

throw new IllegalStateException("Hashtable Enumerator");

if (modCount != expectedModCount)

throw new ConcurrentModificationException();

synchronized(Hashtable.this) {

Entry[] tab = Hashtable.this.table;

int index = (lastReturned.hash & 0x7FFFFFFF) % tab.length;

for (Entry<K,V> e = tab[index], prev = null; e != null;

prev = e, e = e.next) {

if (e == lastReturned) {

modCount++;

expectedModCount++;

if (prev == null)

tab[index] = e.next;

else

prev.next = e.next;

count--;

lastReturned = null;

return;

}

}

throw new ConcurrentModificationException();

}

}

}

private static Enumeration emptyEnumerator = new EmptyEnumerator();

private static Iterator emptyIterator = new EmptyIterator();

/\*\*

\* A hashtable enumerator class for empty hash tables, specializes

\* the general Enumerator

\*/

private static class EmptyEnumerator implements Enumeration<Object> {

EmptyEnumerator() {

}

public boolean hasMoreElements() {

return false;

}

public Object nextElement() {

throw new NoSuchElementException("Hashtable Enumerator");

}

}

/\*\*

\* A hashtable iterator class for empty hash tables

\*/

private static class EmptyIterator implements Iterator<Object> {

EmptyIterator() {

}

public boolean hasNext() {

return false;

}

public Object next() {

throw new NoSuchElementException("Hashtable Iterator");

}

public void remove() {

throw new IllegalStateException("Hashtable Iterator");

}

}

}