**Collections**

**ArrayList** : An ArrayList is the resizable-array implementation of the List interface.

1. ArrayList allows null element to be added.
2. It allows duplicate element to be added.
3. It is fail-fast which indicate that as soon as there is structural modification to the lis, it will fail.
4. It is roughly equivalent to Vector except that it is unsynchronized.
5. Each ArrayList instance has a “capacity”. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.
6. ArrayList can be synchronized using Collections.synchronizedList(new ArrayList(...));
7. Iterator returned by ArrayList are fail-fast: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, 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.
8. ensureCapacity work like below:

int newCapacity = (oldCapacity 3)/2 + 1;

/\*

Replaces the element at the specified position in this list with

the specified element and return the old element.

**@param** index index of the element to replace

**@param** element element to be stored at the specified position

**@return** the element previously at the specified position

**@throws** IndexOutOfBoundsException {@inheritDoc}

\*/

**public** E set(**int** index, E element) {

RangeCheck(index);

E oldValue = (E) elementData[index];

elementData[index] = element;

**return** oldValue;

}

/\*\*

Appends the specified element to the end of this list.

**@param** e element to be appended to this list

**@return** <tt>true</tt> (as specified by {@link Collection#add})

\* /

**public** **boolean** add(E e) {

ensureCapacity(size + 1); // Increments modCount!!

elementData[size++] = e;

**return** **true**;

}

/\*\*

Increases the capacity of this <tt>ArrayList</tt> instance, if

necessary, to ensure that it can hold at least the number of elements

specified by the minimum capacity argument.

**@param** minCapacity the desired minimum capacity\*\* /

**public** **void** ensureCapacity(**int** minCapacity) {

modCount++;

**int** oldCapacity = elementData.length;

**if** (minCapacity > oldCapacity) {

Object oldData[] = elementData;

**int** newCapacity = (oldCapacity 3)/2 + 1;

**if** (newCapacity < minCapacity)

newCapacity = minCapacity;

// minCapacity is usually close to size, so this is a win:

elementData = Arrays.*copyOf*(elementData, newCapacity);

}

}

/\*\*

Removes the first occurrence of the specified element from this list,

if it is present. If the list does not contain the element, it is

unchanged. More formally, removes the element with the lowest index

<tt>i</tt> such that

<tt>(o==null&nbsp;?&nbsp;get(i)==null&nbsp;:&nbsp;o.equals(get(i)))</tt>

(if such an element exists). Returns <tt>true</tt> if this list

contained the specified element (or equivalently, if this list

changed as a result of the call).

\*/

**public** **boolean** remove(Object o) {

**if** (o == **null**) {

**for** (**int** index = 0; index < size; index++)

**if** (elementData[index] == **null**) {

fastRemove(index);

**return** **true**;

}

} **else** {

**for** (**int** index = 0; index < size; index++)

**if** (o.equals(elementData[index])) {

fastRemove(index);

**return** **true**;

}

}

**return** **false**;

}

/\*\*

Appends all of the elements in the specified collection to the end of

this list, in the order that they are returned by the

specified collection's Iterator. The behavior of this operation is

undefined if the specified collection is modified while the operation

is in progress. (This implies that the behavior of this call is

undefined if the specified collection is this list, and this

list is nonempty.) \*\*/

**public** **boolean** addAll(Collection<? **extends** E> c) {

Object[] a = c.toArray();

**int** numNew = a.length;

ensureCapacity(size + numNew); // Increments modCount

System.*arraycopy*(a, 0, elementData, size, numNew);

size += numNew;

**return** numNew != 0;

}

**Vector**

1. Vector allows null element to be added.
2. It allows duplicate element to be added.
3. Each Vector instance has a “capacity”. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an Vector, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.
4. Vector is synchronized;
5. Enumeration returned by Vector’s is fails-safe.
6. The Iterators returned by Vector's iterator and listIterator methods are <em>fail-fast</em>: if the Vector is structurally modified at any time after the Iterator is created, in any way except through the Iterator's own remove or add methods, the Iterator will throw a ConcurrentModificationException.
7. Vector size doubled when it is full. ensureCapacity work like below:

**private** **void** ensureCapacityHelper(**int** minCapacity) {

**int** oldCapacity = elementData.length;

**if** (minCapacity > oldCapacity) {

Object[] oldData = elementData;

**int** newCapacity = (capacityIncrement > 0) ?

(oldCapacity + capacityIncrement) : (oldCapacity \* 2);

**if** (newCapacity < minCapacity) {

newCapacity = minCapacity;

}

elementData = Arrays.*copyOf*(elementData, newCapacity);

}

}

**Vector Demo:**

**package** com.test.collections;

**import** java.util.Enumeration;

**import** java.util.Vector;

**public** **class** VectorDemo {

**private** **static** Vector<Emp> *empVector* = **new** Vector<Emp>();

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

**for**(**int** i=0;i<10;i++){

Emp e=**new** Emp("id"+i,"name"+i,"age"+i,"dob"+i);

*addEmp*(e);

}

*displayVectors*();

}

**public** **static** **void** addEmp(Emp emp){

**if**(emp!=**null**){

*empVector*.add(emp);

}

}

**public** **static** **void** displayVectors(){

Enumeration<Emp> empEnum=*empVector*.elements();

**while**(empEnum.hasMoreElements()){

Emp emp=empEnum.nextElement();

System.*out*.println("empId---"+emp.getEmpId());

System.*out*.println("empName--"+emp.getName());

System.*out*.println("Age--"+emp.getAge());

System.*out*.println("Dob--"+emp.getDob());

}

}

}

**ArrayDeques**

**ArrayDeque**: Resizable-array implementation of the {@link Deque} interface. Its internal representation is as array.

1. Arraydeques have no capacity restrictions; they grow as necessary to supportusage.
2. They are not thread-safe; in the absence of externalsynchronization, they do not support concurrent access by multiple threads
3. Null elements are prohibited.
4. This class is likely to be faster than {@link Stack} when used as a stack, and faster than {@link LinkedList}
5. The iterators returned by this class's iterator method are

fail-fast: If the deque is modified at any time after the iteratoris created, in any way except through the iterator's own remove method, the iterator will generally throw a {@link ConcurrentModificationException}. Thus, in the face of concurrentmodification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

1. **Offer method**: is used to Inserts the specified element at the end of this deque. this is equivalent to offerLast(..) method.
2. **Poll method:** Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty
3. **Element method:** Retrieves, but does not remove, the head of the queue represented by this deque. This method differs from {@link #peek peek} only in that it throws an exception if this deque is empty. This method is equivalent to {@link #getFirst}.
4. **Peek method:** Retrieves, but does not remove, the head of the queue represented by this deque, or returns null if this deque is empty. This method is equivalent to {@link #peekFirst}.@return the head of the queue represented by this deque, or null if this deque is empty.
5. **Push method:** Pushes an element onto the stack represented by this deque. In other words, inserts the element at the front of this deque. This method is equivalent to {@link #addFirst}
6. **Pop method:** Pops an element from the stack represented by this deque. In other words, removes and returns the first element of this deque. This method is equivalent to {@link removeFirst()}.
7. **Delete method:** Removes the element at the specified position in the elements array, adjusting head and tail as necessary. This can result in motion of elements backwards or forwards in the array

**import** java.util.ArrayDeque;

**import** java.util.Deque;

**import** java.util.Iterator;

**public** **class** ArraydequeDemo {

**private** **static** Deque<Emp> *empDeque* = **new** ArrayDeque<Emp>();

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

**for**(**int** i=0;i<10;i++){

Emp e=**new** Emp("id"+i,"name"+i,"age"+i,"dob"+i);

*addEmp*(e);

}

Emp emp=*empDeque*.getFirst();

System.*out*.println("empId--"+emp.getEmpId());

System.*out*.println("empName--"+emp.getName());

//Inserts the specified element at the end of this deque. this is equivalent to offerLast(..) method

*empDeque*.offer(emp);

System.*out*.println(*empDeque*.size());

/\*Retrieves and removes the head of the queue represented by this deque

(in other words, the first element of this deque), or returns null if this deque is empty

\*/

*empDeque*.poll();

//displayDeque();

//Pushes an element onto the stack represented by this deque. In other words, inserts the element at the front of this deque

*empDeque*.push(emp);

//Pops an element from the stack represented by this deque. In other words, removes and returns the first element of this deque

*empDeque*.poll();

}

**public** **static** **void** addEmp(Emp emp){

**if**(emp!=**null**){

*empDeque*.add(emp);

}

}

**public** **static** **void** displayDeque(){

Iterator<Emp> it=*empDeque*.iterator();

**while**(it.hasNext()){

Emp emp=it.next();

System.*out*.println("empId---"+emp.getEmpId());

System.*out*.println("empName--"+emp.getName());

System.*out*.println("Age--"+emp.getAge());

System.*out*.println("Dob--"+emp.getDob());

}

}

}

**LinkedList**

Linked list implementation of the List interface.

1. Allows null to be added.
2. Provides uniformly named methods to get remove and insert an element at the beginning and end of the list. These operations allow linked lists to be used as a stack, {@linkplain Queue queue}, or {@linkplain Deque double-ended queue}.
3. It is not thread safe.
4. Iterator returned by this class is fail-fast.
5. Internal representation as LinkList.
6. It implements the Dqueue interface, so all the method of Deque is already implemented.

**package** com.test.collections;

**import** java.util.Deque;

**import** java.util.Iterator;

**import** java.util.LinkedList;

**public** **class** LinkedListDemo {

**private** **static** Deque<Emp> *empDeque* = **new** LinkedList<Emp>();

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

**for**(**int** i=0;i<10;i++){

Emp e=**new** Emp("id"+i,"name"+i,"age"+i,"dob"+i);

*addEmp*(e);

}

Emp emp=*empDeque*.getFirst();

System.*out*.println("empId--"+emp.getEmpId());

System.*out*.println("empName--"+emp.getName());

//Inserts the specified element at the end of this deque. this is equivalent to offerLast(..) method

*empDeque*.offer(emp);

System.*out*.println(*empDeque*.size());

/\*Retrieves and removes the head of the queue represented by this deque

(in other words, the first element of this deque), or returns null if this deque is empty

\*/

*empDeque*.poll();

//displayDeque();

//Pushes an element onto the stack represented by this deque. In other words, inserts the element at the front of this deque

*empDeque*.push(emp);

//Pops an element from the stack represented by this deque. In other words, removes and returns the first element of this deque

*empDeque*.poll();

}

**public** **static** **void** addEmp(Emp emp){

**if**(emp!=**null**){

*empDeque*.add(emp);

}

}

**public** **static** **void** displayDeque(){

Iterator<Emp> it=*empDeque*.iterator();

**while**(it.hasNext()){

Emp emp=it.next();

System.*out*.println("empId---"+emp.getEmpId());

System.*out*.println("empName--"+emp.getName());

System.*out*.println("Age--"+emp.getAge());

System.*out*.println("Dob--"+emp.getDob());

}

}

}

**PriorityQueue**

PriorityQueue: An unbounded priority {@linkplain Queue queue} based on a priority heap.

1. The elements of the priority queue are ordered according to their {@linkplain Comparable natural ordering}, or by a {@link Comparator} provided at queue construction time, depending on which constructor is used.
2. A priority queue does not permit {@code null} elements.
3. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in {@code ClassCastException}).
4. The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily.
5. The queue retrieval operations {@code poll}, {@code remove}, {@code peek}, and {@code element} access the element at the head of the queue.
6. It is not synchronized.
7. Iterator returned by this class is fail-fast.

**EnumMap**

EnumMap is a specialized {@link Map} implementation for use with enum type keys. All of the keys in an enum map must come from a single enum type that is specified, explicitly or implicitly, when the map is created. Enum maps are represented internally as arrays. This representation is extremely compact and efficient.

1. Enum maps are maintained in the <i>natural order</i> of their keys (the order in which the enum constants are declared). This is reflected in the iterators returned by the collections views ({@link #keySet()}, {@link #entrySet()}, and {@link #values()}).
2. Iterators returned by the collection views are <i>weakly consistent</i>:

they will never throw {@link ConcurrentModificationException} and they may or may not show the effects of any modifications to the map that occur while the iteration is in progress.

1. Null keys are not permitted. Attempts to insert a null key will throw {@link NullPointerException}. Attempts to test for the presence of a null key or to remove one will, however, function properly.
2. Null values are permitted.
3. EnumMap is not synchronized. If multiple threads access an enum map concurrently, and at least one of the threads modifies the map, it should be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the enum map. If no such object exists, the map should be "wrapped" using the {@link Collections#synchronizedMap} method. This is best done at creation time, to prevent accidental unsynchronized access:
4. All basic operations execute in constant time. They are likely (though not guaranteed) to be faster than their {@link HashMap} counterparts.
5. For each loop cannot be used with EnumMap.

**package** com.test.collections;

**import** java.util.EnumMap;

**import** java.util.Iterator;

/

EnumMap is a specialized {@link Map} implementation for use with enum type

keys. All of the keys in an enum map must come from a single enum type that

is specified, explicitly or implicitly, when the map is created. Enum maps

are represented internally as arrays. This representation is extremely

compact and efficient.

i) Enum maps are maintained in the <i>natural

order</i> of their keys (the order in which the enum constants are declared).

This is reflected in the iterators returned by the collections views (

{@link #keySet()}, {@link #entrySet()}, and {@link #values()}).

ii) Iterators returned by the collection views are <i>weakly consistent</i>: they will

never throw {@link ConcurrentModificationException} and they may or may not

show the effects of any modifications to the map that occur while the

iteration is in progress.

iii) Null keys are not permitted. Attempts to insert a null key will throw

{@link NullPointerException}. Attempts to test for the presence of a null key

or to remove one will, however, function properly. iv) Null values are

permitted. v) EnumMap is not synchronized. If multiple threads access an enum

map concurrently, and at least one of the threads modifies the map, it should

be synchronized externally. This is typically accomplished by synchronizing

on some object that naturally encapsulates the enum map. If no such object

exists, the map should be "wrapped" using the

{@link Collections#synchronizedMap} method. This is best done at creation

time, to prevent accidental unsynchronized access: vi) All basic operations

execute in constant time. They are likely (though not guaranteed) to be

faster than their {@link HashMap} counterparts.

**@author** rakeshku

/

**public** **class** EnumMapDemo {

**public** **enum** Number {

*ONE*, *TWO*, *THREE*

}

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

EnumMap<Number, String> em = **new** EnumMap<Number, String>(Number.**class**);

em.put(Number.*ONE*, "1");

em.put(Number.*TWO*, "2");

em.put(Number.*THREE*, "3");

Iterator<Number> it = em.keySet().iterator();

**while** (it.hasNext()) {

System.*out*.println("---" + it.next());

}

**while** (it.hasNext()) {

System.*out*.println("---" + it.next());

}

}

}

**package** java.util;

**import** java.util.Map.Entry;

**import** sun.misc.SharedSecrets;

/

A specialized {@link Map} implementation for use with enum type keys. All

of the keys in an enum map must come from a single enum type that is

specified, explicitly or implicitly, when the map is created. Enum maps

are represented internally as arrays. This representation is extremely

compact and efficient.

<p>Enum maps are maintained in the <i>natural order</i> of their keys

(the order in which the enum constants are declared). This is reflected

in the iterators returned by the collections views ({@link #keySet()},

{@link #entrySet()}, and {@link #values()}).

<p>Iterators returned by the collection views are <i>weakly consistent</i>:

they will never throw {@link ConcurrentModificationException} and they may

or may not show the effects of any modifications to the map that occur while

the iteration is in progress.

<p>Null keys are not permitted. Attempts to insert a null key will

throw {@link NullPointerException}. Attempts to test for the

presence of a null key or to remove one will, however, function properly.

Null values are permitted.

<P>Like most collection implementations <tt>EnumMap</tt> is not

synchronized. If multiple threads access an enum map concurrently, and at

least one of the threads modifies the map, it should be synchronized

externally. This is typically accomplished by synchronizing on some

object that naturally encapsulates the enum map. If no such object exists,

the map should be "wrapped" using the {@link Collections#synchronizedMap}

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

unsynchronized access:

<pre>

Map&lt;EnumKey, V&gt; m

= Collections.synchronizedMap(new EnumMap&lt;EnumKey, V&gt;(...));

</pre>

<p>Implementation note: All basic operations execute in constant time.

They are likely (though not guaranteed) to be faster than their

{@link HashMap} counterparts.

<p>This class is a member of the

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

Java Collections Framework</a>.

**@author** Josh Bloch

**@version** %I%, %G%

**@see** EnumSet

**@since** 1.5

/

**public** **class** EnumMap<K **extends** Enum<K>, V> **extends** AbstractMap<K, V>

**implements** java.io.Serializable, Cloneable

{

/

The <tt>Class</tt> object for the enum type of all the keys of this map.

**@serial**

/

**private** **final** Class<K> keyType;

/

All of the values comprising K. (Cached for performance.)

/

**private** **transient** K[] keyUniverse;

/

Array representation of this map. The ith element is the value

to which universe[i] is currently mapped, or null if it isn't

mapped to anything, or NULL if it's mapped to null.

/

**private** **transient** Object[] vals;

/

The number of mappings in this map.

/

**private** **transient** **int** size = 0;

/

Distinguished non-null value for representing null values.

/

**private** **static** **final** Object *NULL* = **new** Object();

**private** Object maskNull(Object value) {

**return** (value == **null** ? *NULL* : value);

}

**private** V unmaskNull(Object value) {

**return** (V) (value == *NULL* ? **null** : value);

}

**private** **static** Enum[] *ZERO\_LENGTH\_ENUM\_ARRAY* = **new** Enum[0];

/

Creates an empty enum map with the specified key type.

**@param** keyType the class object of the key type for this enum map

**@throws** NullPointerException if <tt>keyType</tt> is null

/

**public** EnumMap(Class<K> keyType) {

**this**.keyType = keyType;

keyUniverse = *getKeyUniverse*(keyType);

vals = **new** Object[keyUniverse.length];

}

/

Creates an enum map with the same key type as the specified enum

map, initially containing the same mappings (if any).

**@param** m the enum map from which to initialize this enum map

**@throws** NullPointerException if <tt>m</tt> is null

/

**public** EnumMap(EnumMap<K, ? **extends** V> m) {

keyType = m.keyType;

keyUniverse = m.keyUniverse;

vals = (Object[]) m.vals.clone();

size = m.size;

}

/

Creates an enum map initialized from the specified map. If the

specified map is an <tt>EnumMap</tt> instance, this constructor behaves

identically to {@link #EnumMap(EnumMap)}. Otherwise, the specified map

must contain at least one mapping (in order to determine the new

enum map's key type).

**@param** m the map from which to initialize this enum map

**@throws** IllegalArgumentException if <tt>m</tt> is not an

<tt>EnumMap</tt> instance and contains no mappings

**@throws** NullPointerException if <tt>m</tt> is null

/

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

**if** (m **instanceof** EnumMap) {

EnumMap<K, ? **extends** V> em = (EnumMap<K, ? **extends** V>) m;

keyType = em.keyType;

keyUniverse = em.keyUniverse;

vals = (Object[]) em.vals.clone();

size = em.size;

} **else** {

**if** (m.isEmpty())

**throw** **new** IllegalArgumentException("Specified map is empty");

keyType = m.keySet().iterator().next().getDeclaringClass();

keyUniverse = *getKeyUniverse*(keyType);

vals = **new** Object[keyUniverse.length];

putAll(m);

}

}

// Query Operations

/

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 maps one or more keys to the

specified value.

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

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

/

**public** **boolean** containsValue(Object value) {

value = maskNull(value);

**for** (Object val : vals)

**if** (value.equals(val))

**return** **true**;

**return** **false**;

}

/

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** isValidKey(key) && vals[((Enum)key).ordinal()] != **null**;

}

**private** **boolean** containsMapping(Object key, Object value) {

**return** isValidKey(key) &&

maskNull(value).equals(vals[((Enum)key).ordinal()]);

}

/

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 == 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.

/

**public** V get(Object key) {

**return** (isValidKey(key) ?

unmaskNull(vals[((Enum)key).ordinal()]) : **null**);

}

// Modification Operations

/

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

If the map previously contained a mapping for this key, the old

value is replaced.

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

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

**@return** the previous value associated with specified key, or

<tt>null</tt> if there was no mapping for key. (A <tt>null</tt>

return can also indicate that the map previously associated

<tt>null</tt> with the specified key.)

**@throws** NullPointerException if the specified key is null

/

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

typeCheck(key);

**int** index = ((Enum)key).ordinal();

Object oldValue = vals[index];

vals[index] = maskNull(value);

**if** (oldValue == **null**)

size++;

**return** unmaskNull(oldValue);

}

/

Removes the mapping for this key from this map if present.

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

**@return** the previous value associated with specified key, or

<tt>null</tt> if there was no entry for key. (A <tt>null</tt>

return can also indicate that the map previously associated

<tt>null</tt> with the specified key.)

/

**public** V remove(Object key) {

**if** (!isValidKey(key))

**return** **null**;

**int** index = ((Enum)key).ordinal();

Object oldValue = vals[index];

vals[index] = **null**;

**if** (oldValue != **null**)

size--;

**return** unmaskNull(oldValue);

}

**private** **boolean** removeMapping(Object key, Object value) {

**if** (!isValidKey(key))

**return** **false**;

**int** index = ((Enum)key).ordinal();

**if** (maskNull(value).equals(vals[index])) {

vals[index] = **null**;

size--;

**return** **true**;

}

**return** **false**;

}

/

Returns true if key is of the proper type to be a key in this

enum map.

/

**private** **boolean** isValidKey(Object key) {

**if** (key == **null**)

**return** **false**;

// Cheaper than instanceof Enum followed by getDeclaringClass

Class keyClass = key.getClass();

**return** keyClass == keyType || keyClass.getSuperclass() == keyType;

}

// Bulk Operations

/

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 the mappings to be stored in this map

**@throws** NullPointerException the specified map is null, or if

one or more keys in the specified map are null

/

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

**if** (m **instanceof** EnumMap) {

EnumMap<? **extends** K, ? **extends** V> em =

(EnumMap<? **extends** K, ? **extends** V>)m;

**if** (em.keyType != keyType) {

**if** (em.isEmpty())

**return**;

**throw** **new** ClassCastException(em.keyType + " != " + keyType);

}

**for** (**int** i = 0; i < keyUniverse.length; i++) {

Object emValue = em.vals[i];

**if** (emValue != **null**) {

**if** (vals[i] == **null**)

size++;

vals[i] = emValue;

}

}

} **else** {

**super**.putAll(m);

}

}

/

Removes all mappings from this map.

/

**public** **void** clear() {

Arrays.*fill*(vals, **null**);

size = 0;

}

// Views

/

This field is initialized to contain an instance of the entry set

view the first time this view is requested. The view is stateless,

so there's no reason to create more than one.

/

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

/

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

The returned set obeys the general contract outlined in

{@link Map#keySet()}. The set's iterator will return the keys

in their natural order (the order in which the enum constants

are declared).

**@return** a set view of the keys contained in this enum map

/

**public** Set<K> keySet() {

Set<K> ks = keySet;

**if** (ks != **null**)

**return** ks;

**else**

**return** keySet = **new** KeySet();

}

**private** **class** KeySet **extends** AbstractSet<K> {

**public** Iterator<K> iterator() {

**return** **new** KeyIterator();

}

**public** **int** size() {

**return** size;

}

**public** **boolean** contains(Object o) {

**return** containsKey(o);

}

**public** **boolean** remove(Object o) {

**int** oldSize = size;

EnumMap.**this**.remove(o);

**return** size != oldSize;

}

**public** **void** clear() {

EnumMap.**this**.clear();

}

}

/

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

The returned collection obeys the general contract outlined in

{@link Map#values()}. The collection's iterator will return the

values in the order their corresponding keys appear in map,

which is their natural order (the order in which the enum constants

are declared).

**@return** a collection view of the values contained in this map

/

**public** Collection<V> values() {

Collection<V> vs = values;

**if** (vs != **null**)

**return** vs;

**else**

**return** values = **new** Values();

}

**private** **class** Values **extends** AbstractCollection<V> {

**public** Iterator<V> iterator() {

**return** **new** ValueIterator();

}

**public** **int** size() {

**return** size;

}

**public** **boolean** contains(Object o) {

**return** containsValue(o);

}

**public** **boolean** remove(Object o) {

o = maskNull(o);

**for** (**int** i = 0; i < vals.length; i++) {

**if** (o.equals(vals[i])) {

vals[i] = **null**;

size--;

**return** **true**;

}

}

**return** **false**;

}

**public** **void** clear() {

EnumMap.**this**.clear();

}

}

/

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

The returned set obeys the general contract outlined in

{@link Map#keySet()}. The set's iterator will return the

mappings in the order their keys appear in map, which is their

natural order (the order in which the enum constants are declared).

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

/

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

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

**if** (es != **null**)

**return** es;

**else**

**return** entrySet = **new** EntrySet();

}

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

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

**return** **new** EntryIterator();

}

**public** **boolean** contains(Object o) {

**if** (!(o **instanceof** Map.Entry))

**return** **false**;

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

**return** containsMapping(entry.getKey(), entry.getValue());

}

**public** **boolean** remove(Object o) {

**if** (!(o **instanceof** Map.Entry))

**return** **false**;

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

**return** removeMapping(entry.getKey(), entry.getValue());

}

**public** **int** size() {

**return** size;

}

**public** **void** clear() {

EnumMap.**this**.clear();

}

**public** Object[] toArray() {

**return** fillEntryArray(**new** Object[size]);

}

@SuppressWarnings("unchecked")

**public** <T> T[] toArray(T[] a) {

**int** size = size();

**if** (a.length < size)

a = (T[])java.lang.reflect.Array

.*newInstance*(a.getClass().getComponentType(), size);

**if** (a.length > size)

a[size] = **null**;

**return** (T[]) fillEntryArray(a);

}

**private** Object[] fillEntryArray(Object[] a) {

**int** j = 0;

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

**if** (vals[i] != **null**)

a[j++] = **new** AbstractMap.SimpleEntry<K,V>(

keyUniverse[i], unmaskNull(vals[i]));

**return** a;

}

}

**private** **abstract** **class** EnumMapIterator<T> **implements** Iterator<T> {

// Lower bound on index of next element to return

**int** index = 0;

// Index of last returned element, or -1 if none

**int** lastReturnedIndex = -1;

**public** **boolean** hasNext() {

**while** (index < vals.length && vals[index] == **null**)

index++;

**return** index != vals.length;

}

**public** **void** remove() {

checkLastReturnedIndex();

**if** (vals[lastReturnedIndex] != **null**) {

vals[lastReturnedIndex] = **null**;

size--;

}

lastReturnedIndex = -1;

}

**private** **void** checkLastReturnedIndex() {

**if** (lastReturnedIndex < 0)

**throw** **new** IllegalStateException();

}

}

**private** **class** KeyIterator **extends** EnumMapIterator<K> {

**public** K next() {

**if** (!hasNext())

**throw** **new** NoSuchElementException();

lastReturnedIndex = index++;

**return** keyUniverse[lastReturnedIndex];

}

}

**private** **class** ValueIterator **extends** EnumMapIterator<V> {

**public** V next() {

**if** (!hasNext())

**throw** **new** NoSuchElementException();

lastReturnedIndex = index++;

**return** unmaskNull(vals[lastReturnedIndex]);

}

}

/

Since we don't use Entry objects, we use the Iterator itself as entry.

/

**private** **class** EntryIterator **extends** EnumMapIterator<Map.Entry<K,V>>

**implements** Map.Entry<K,V>

{

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

**if** (!hasNext())

**throw** **new** NoSuchElementException();

lastReturnedIndex = index++;

**return** **this**;

}

**public** K getKey() {

checkLastReturnedIndexForEntryUse();

**return** keyUniverse[lastReturnedIndex];

}

**public** V getValue() {

checkLastReturnedIndexForEntryUse();

**return** unmaskNull(vals[lastReturnedIndex]);

}

**public** V setValue(V value) {

checkLastReturnedIndexForEntryUse();

V oldValue = unmaskNull(vals[lastReturnedIndex]);

vals[lastReturnedIndex] = maskNull(value);

**return** oldValue;

}

**public** **boolean** equals(Object o) {

**if** (lastReturnedIndex < 0)

**return** o == **this**;

**if** (!(o **instanceof** Map.Entry))

**return** **false**;

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

V ourValue = unmaskNull(vals[lastReturnedIndex]);

Object hisValue = e.getValue();

**return** e.getKey() == keyUniverse[lastReturnedIndex] &&

(ourValue == hisValue ||

(ourValue != **null** && ourValue.equals(hisValue)));

}

**public** **int** hashCode() {

**if** (lastReturnedIndex < 0)

**return** **super**.hashCode();

Object value = vals[lastReturnedIndex];

**return** keyUniverse[lastReturnedIndex].hashCode()

^ (value == *NULL* ? 0 : value.hashCode());

}

**public** String toString() {

**if** (lastReturnedIndex < 0)

**return** **super**.toString();

**return** keyUniverse[lastReturnedIndex] + "="

+ unmaskNull(vals[lastReturnedIndex]);

}

**private** **void** checkLastReturnedIndexForEntryUse() {

**if** (lastReturnedIndex < 0)

**throw** **new** IllegalStateException("Entry was removed");

}

}

// Comparison and hashing

/

Compares the specified object with this map for equality. Returns

<tt>true</tt> if the given object is also a map and the two maps

represent the same mappings, as specified in the {@link

Map#equals(Object)} contract.

**@param** o the object to be compared for equality with this map

**@return** <tt>true</tt> if the specified object is equal to this map

/

**public** **boolean** equals(Object o) {

**if** (!(o **instanceof** EnumMap))

**return** **super**.equals(o);

EnumMap em = (EnumMap)o;

**if** (em.keyType != keyType)

**return** size == 0 && em.size == 0;

// Key types match, compare each value

**for** (**int** i = 0; i < keyUniverse.length; i++) {

Object ourValue = vals[i];

Object hisValue = em.vals[i];

**if** (hisValue != ourValue &&

(hisValue == **null** || !hisValue.equals(ourValue)))

**return** **false**;

}

**return** **true**;

}

/

Returns a shallow copy of this enum map. (The values themselves

are not cloned.

**@return** a shallow copy of this enum map

/

**public** EnumMap<K, V> clone() {

EnumMap<K, V> result = **null**;

**try** {

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

} **catch**(CloneNotSupportedException e) {

**throw** **new** AssertionError();

}

result.vals = (Object[]) result.vals.clone();

**return** result;

}

/

Throws an exception if e is not of the correct type for this enum set.

/

**private** **void** typeCheck(K key) {

Class keyClass = key.getClass();

**if** (keyClass != keyType && keyClass.getSuperclass() != keyType)

**throw** **new** ClassCastException(keyClass + " != " + keyType);

}

/

Returns all of the values comprising K.

The result is uncloned, cached, and shared by all callers.

/

**private** **static** <K **extends** Enum<K>> K[] getKeyUniverse(Class<K> keyType) {

**return** SharedSecrets.*getJavaLangAccess*()

.getEnumConstantsShared(keyType);

}

**private** **static** **final** **long** *serialVersionUID* = 458661240069192865L;

/

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

serialize it).

**@serialData** The <i>size</i> of the enum map (the number of key-value

mappings) is emitted (int), followed by the key (Object)

and value (Object) for each key-value mapping represented

by the enum map.

/

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

**throws** java.io.IOException

{

// Write out the key type and any hidden stuff

s.defaultWriteObject();

// Write out size (number of Mappings)

s.writeInt(size);

// Write out keys and values (alternating)

**for** (Map.Entry<K,V> e : entrySet()) {

s.writeObject(e.getKey());

s.writeObject(e.getValue());

}

}

/

Reconstitute the <tt>EnumMap</tt> instance from a stream (i.e.,

deserialize it).

/

**private** **void** readObject(java.io.ObjectInputStream s)

**throws** java.io.IOException, ClassNotFoundException

{

// Read in the key type and any hidden stuff

s.defaultReadObject();

keyUniverse = *getKeyUniverse*(keyType);

vals = **new** Object[keyUniverse.length];

// 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();

put(key, value);

}

}

}

**EnumSet**

EnumSet is a specialized {@link Set} implementation for use with enum types. All of the elements in an enum set must come from a single enum type that is specified, explicitly or implicitly, when the set is created. Enum sets are represented internally as bit vectors. This representation is extremely compact and efficient. The space and time performance of this class should be good enough to allow its use as a high-quality, typesafe alternative to traditional int-based "bit flags." Even bulk operations (such as containsAll and retainAll) should run very quickly if their argument is also an enum set.

1. The iterator returned by the iterator method traverses the elements in their <i>natural order</i> (the order in which the enum constants are declared). The returned iterator is <i>weakly consistent</i>: it will never throw {@link ConcurrentModificationException} and it may or may not show the effects of any modifications to the set that occur while the iteration is in progress
2. Null elements are not permitted. Attempts to insert a null element will throw {@link NullPointerException}. Attempts to test for the presence of a null element or to remove one will, however, function properly.
3. EnumSet is not synchronized.
4. All basic operations execute in constant time. They are likely (though not guaranteed) to be faster than their {@link HashSet} counterparts.

**public** **abstract** **class** EnumSet<E **extends** Enum<E>> **extends** AbstractSet<E>

**implements** Cloneable, java.io.Serializable

{

/\*\*

\* The class of all the elements of this set.

\*/

**final** Class<E> elementType;

/\*\*

\* All of the values comprising T. (Cached for performance.)

\*/

**final** Enum[] universe;

**private** **static** Enum[] *ZERO\_LENGTH\_ENUM\_ARRAY* = **new** Enum[0];

EnumSet(Class<E>elementType, Enum[] universe) {

**this**.elementType = elementType;

**this**.universe = universe;

}

/\*\*

\* Creates an empty enum set with the specified element type.

\*

\* **@param** elementType the class object of the element type for this enum

\* set

\* **@throws** NullPointerException if <tt>elementType</tt> is null

\*/

**public** **static** <E **extends** Enum<E>> EnumSet<E> noneOf(Class<E> elementType) {

Enum[] universe = *getUniverse*(elementType);

**if** (universe == **null**)

**throw** **new** ClassCastException(elementType + " not an enum");

**if** (universe.length <= 64)

**return** **new** RegularEnumSet<E>(elementType, universe);

**else**

**return** **new** JumboEnumSet<E>(elementType, universe);

}

/\*\*

\* Creates an enum set containing all of the elements in the specified

\* element type.

\*

\* **@param** elementType the class object of the element type for this enum

\* set

\* **@throws** NullPointerException if <tt>elementType</tt> is null

\*/

**public** **static** <E **extends** Enum<E>> EnumSet<E> allOf(Class<E> elementType) {

EnumSet<E> result = *noneOf*(elementType);

result.addAll();

**return** result;

}

/\*\*

\* Adds all of the elements from the appropriate enum type to this enum

\* set, which is empty prior to the call.

\*/

**abstract** **void** addAll();

/\*\*

\* Creates an enum set with the same element type as the specified enum

\* set, initially containing the same elements (if any).

\*

\* **@param** s the enum set from which to initialize this enum set

\* **@throws** NullPointerException if <tt>s</tt> is null

\*/

**public** **static** <E **extends** Enum<E>> EnumSet<E> copyOf(EnumSet<E> s) {

**return** s.clone();

}

/\*\*

\* Creates an enum set initialized from the specified collection. If

\* the specified collection is an <tt>EnumSet</tt> instance, this static

\* factory method behaves identically to {@link #copyOf(EnumSet)}.

\* Otherwise, the specified collection must contain at least one element

\* (in order to determine the new enum set's element type).

\*

\* **@param** c the collection from which to initialize this enum set

\* **@throws** IllegalArgumentException if <tt>c</tt> is not an

\* <tt>EnumSet</tt> instance and contains no elements

\* **@throws** NullPointerException if <tt>c</tt> is null

\*/

**public** **static** <E **extends** Enum<E>> EnumSet<E> copyOf(Collection<E> c) {

**if** (c **instanceof** EnumSet) {

**return** ((EnumSet<E>)c).clone();

} **else** {

**if** (c.isEmpty())

**throw** **new** IllegalArgumentException("Collection is empty");

Iterator<E> i = c.iterator();

E first = i.next();

EnumSet<E> result = EnumSet.*of*(first);

**while** (i.hasNext())

result.add(i.next());

**return** result;

}

}

/\*\*

\* Creates an enum set with the same element type as the specified enum

\* set, initially containing all the elements of this type that are

\* <i>not</i> contained in the specified set.

\*

\* **@param** s the enum set from whose complement to initialize this enum set

\* **@throws** NullPointerException if <tt>s</tt> is null

\*/

**public** **static** <E **extends** Enum<E>> EnumSet<E> complementOf(EnumSet<E> s) {

EnumSet<E> result = *copyOf*(s);

result.complement();

**return** result;

}

/\*\*

\* Creates an enum set initially containing the specified element.

\*

\* Overloadings of this method exist to initialize an enum set with

\* one through five elements. A sixth overloading is provided that

\* uses the varargs feature. This overloading may be used to create

\* an enum set initially containing an arbitrary number of elements, but

\* is likely to run slower than the overloadings that do not use varargs.

\*

\* **@param** e the element that this set is to contain initially

\* **@throws** NullPointerException if <tt>e</tt> is null

\* **@return** an enum set initially containing the specified element

\*/

**public** **static** <E **extends** Enum<E>> EnumSet<E> of(E e) {

EnumSet<E> result = *noneOf*(e.getDeclaringClass());

result.add(e);

**return** result;

}

/\*\*

\* Creates an enum set initially containing the specified elements.

\*

\* Overloadings of this method exist to initialize an enum set with

\* one through five elements. A sixth overloading is provided that

\* uses the varargs feature. This overloading may be used to create

\* an enum set initially containing an arbitrary number of elements, but

\* is likely to run slower than the overloadings that do not use varargs.

\*

\* **@param** e1 an element that this set is to contain initially

\* **@param** e2 another element that this set is to contain initially

\* **@throws** NullPointerException if any parameters are null

\* **@return** an enum set initially containing the specified elements

\*/

**public** **static** <E **extends** Enum<E>> EnumSet<E> of(E e1, E e2) {

EnumSet<E> result = *noneOf*(e1.getDeclaringClass());

result.add(e1);

result.add(e2);

**return** result;

}

/\*\*

\* Creates an enum set initially containing the specified elements.

\*

\* Overloadings of this method exist to initialize an enum set with

\* one through five elements. A sixth overloading is provided that

\* uses the varargs feature. This overloading may be used to create

\* an enum set initially containing an arbitrary number of elements, but

\* is likely to run slower than the overloadings that do not use varargs.

\*

\* **@param** e1 an element that this set is to contain initially

\* **@param** e2 another element that this set is to contain initially

\* **@param** e3 another element that this set is to contain initially

\* **@throws** NullPointerException if any parameters are null

\* **@return** an enum set initially containing the specified elements

\*/

**public** **static** <E **extends** Enum<E>> EnumSet<E> of(E e1, E e2, E e3) {

EnumSet<E> result = *noneOf*(e1.getDeclaringClass());

result.add(e1);

result.add(e2);

result.add(e3);

**return** result;

}

/\*\*

\* Creates an enum set initially containing the specified elements.

\*

\* Overloadings of this method exist to initialize an enum set with

\* one through five elements. A sixth overloading is provided that

\* uses the varargs feature. This overloading may be used to create

\* an enum set initially containing an arbitrary number of elements, but

\* is likely to run slower than the overloadings that do not use varargs.

\*

\* **@param** e1 an element that this set is to contain initially

\* **@param** e2 another element that this set is to contain initially

\* **@param** e3 another element that this set is to contain initially

\* **@param** e4 another element that this set is to contain initially

\* **@throws** NullPointerException if any parameters are null

\* **@return** an enum set initially containing the specified elements

\*/

**public** **static** <E **extends** Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4) {

EnumSet<E> result = *noneOf*(e1.getDeclaringClass());

result.add(e1);

result.add(e2);

result.add(e3);

result.add(e4);

**return** result;

}

/\*\*

\* Creates an enum set initially containing the specified elements.

\*

\* Overloadings of this method exist to initialize an enum set with

\* one through five elements. A sixth overloading is provided that

\* uses the varargs feature. This overloading may be used to create

\* an enum set initially containing an arbitrary number of elements, but

\* is likely to run slower than the overloadings that do not use varargs.

\*

\* **@param** e1 an element that this set is to contain initially

\* **@param** e2 another element that this set is to contain initially

\* **@param** e3 another element that this set is to contain initially

\* **@param** e4 another element that this set is to contain initially

\* **@param** e5 another element that this set is to contain initially

\* **@throws** NullPointerException if any parameters are null

\* **@return** an enum set initially containing the specified elements

\*/

**public** **static** <E **extends** Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4,

E e5)

{

EnumSet<E> result = *noneOf*(e1.getDeclaringClass());

result.add(e1);

result.add(e2);

result.add(e3);

result.add(e4);

result.add(e5);

**return** result;

}

/\*\*

\* Creates an enum set initially containing the specified elements.

\* This factory, whose parameter list uses the varargs feature, may

\* be used to create an enum set initially containing an arbitrary

\* number of elements, but it is likely to run slower than the overloadings

\* that do not use varargs.

\*

\* **@param** first an element that the set is to contain initially

\* **@param** rest the remaining elements the set is to contain initially

\* **@throws** NullPointerException if any of the specified elements are null,

\* or if <tt>rest</tt> is null

\* **@return** an enum set initially containing the specified elements

\*/

**public** **static** <E **extends** Enum<E>> EnumSet<E> of(E first, E... rest) {

EnumSet<E> result = *noneOf*(first.getDeclaringClass());

result.add(first);

**for** (E e : rest)

result.add(e);

**return** result;

}

/\*\*

\* Creates an enum set initially containing all of the elements in the

\* range defined by the two specified endpoints. The returned set will

\* contain the endpoints themselves, which may be identical but must not

\* be out of order.

\*

\* **@param** from the first element in the range

\* **@param** to the last element in the range

\* **@throws** NullPointerException if <tt>first</tt> or <tt>last</tt> are

\* null

\* **@throws** IllegalArgumentException if <tt>first.compareTo(last) &gt; 0</tt>

\* **@return** an enum set initially containing all of the elements in the

\* range defined by the two specified endpoints

\*/

**public** **static** <E **extends** Enum<E>> EnumSet<E> range(E from, E to) {

**if** (from.compareTo(to) > 0)

**throw** **new** IllegalArgumentException(from + " > " + to);

EnumSet<E> result = *noneOf*(from.getDeclaringClass());

result.addRange(from, to);

**return** result;

}

/\*\*

\* Adds the specified range to this enum set, which is empty prior

\* to the call.

\*/

**abstract** **void** addRange(E from, E to);

/\*\*

\* Returns a copy of this set.

\*

\* **@return** a copy of this set

\*/

**public** EnumSet<E> clone() {

**try** {

**return** (EnumSet<E>) **super**.clone();

} **catch**(CloneNotSupportedException e) {

**throw** **new** AssertionError(e);

}

}

/\*\*

\* Complements the contents of this enum set.

\*/

**abstract** **void** complement();

/\*\*

\* Throws an exception if e is not of the correct type for this enum set.

\*/

**final** **void** typeCheck(E e) {

Class eClass = e.getClass();

**if** (eClass != elementType && eClass.getSuperclass() != elementType)

**throw** **new** ClassCastException(eClass + " != " + elementType);

}

/\*\*

\* Returns all of the values comprising E.

\* The result is uncloned, cached, and shared by all callers.

\*/

**private** **static** <E **extends** Enum<E>> E[] getUniverse(Class<E> elementType) {

**return** SharedSecrets.*getJavaLangAccess*()

.getEnumConstantsShared(elementType);

}

/\*\*

\* This class is used to serialize all EnumSet instances, regardless of

\* implementation type. It captures their "logical contents" and they

\* are reconstructed using public static factories. This is necessary

\* to ensure that the existence of a particular implementation type is

\* an implementation detail.

\*

\* **@serial** include

\*/

**private** **static** **class** SerializationProxy <E **extends** Enum<E>>

**implements** java.io.Serializable

{

/\*\*

\* The element type of this enum set.

\*

\* **@serial**

\*/

**private** **final** Class<E> elementType;

/\*\*

\* The elements contained in this enum set.

\*

\* **@serial**

\*/

**private** **final** Enum[] elements;

SerializationProxy(EnumSet<E> set) {

elementType = set.elementType;

elements = (Enum[]) set.toArray(*ZERO\_LENGTH\_ENUM\_ARRAY*);

}

**private** Object readResolve() {

EnumSet<E> result = EnumSet.*noneOf*(elementType);

**for** (Enum e : elements)

result.add((E)e);

**return** result;

}

**private** **static** **final** **long** *serialVersionUID* = 362491234563181265L;

}

Object writeReplace() {

**return** **new** SerializationProxy<E>(**this**);

}

}

**BlockingQueue**

## BlockingQueue Usage

A BlockingQueue is typically used to have on thread produce objects, which another thread consumes. Here is a diagram that illustrates this principle:



**A BlockingQueue with one thread putting into it, and another thread taking from it.**

The producing thread will keep producing new objects and insert them into the queue, until the queue reaches some upper bound on what it can contain. It's limit, in other words. If the blocking queue reaches its upper limit, the producing thread is blocked while trying to insert the new object. It remains blocked until a consuming thread takes an object out of the queue.

The consuming thread keeps taking objects out of the blocking queue, and processes them. If the consuming thread tries to take an object out of an empty queue, the consuming thread is blocked until a producing thread puts an object into the queue.

### BlockingQueue Methods

A BlockingQueue has 4 different sets of methods for inserting, removing and examining the elements in the queue. Each set of methods behaves differently in case the requested operation cannot be carried out immediately. Here is a table of the methods:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws Exception** | **Special Value** | **Blocks** | **Times Out** |
| **Insert** | add(o) | offer(o) | put(o) | offer(o, timeout, timeunit) |
| **Remove** | remove(o) | poll() | take() | poll(timeout, timeunit) |
| **Examine** | element() | peek() |  |  |

The 4 different sets of behavior means this:

1. **Throws Exception**:   
   If the attempted operation is not possible immediately, an exception is thrown.
2. **Special Value**:   
   If the attempted operation is not possible immediately, a special value is returned (often true / false).
3. **Blocks**:   
   If the attempted operation is not possible immediately, the method call blocks until it is.
4. **Times Out**:   
   If the attempted operation is not possible immediately, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).

It is not possible to insert null into a **BlockingQueue**. If you try to insert null, the **BlockingQueue** will throw a NullPointerException.

It is also possible to access all the elements inside a **BlockingQueue**, and not just the elements at the start and end. For instance, say you have queued an object for processing, but your application decides to cancel it. You can then call e.g. remove(o) to remove a specific object in the queue. However, this is not done very efficiently, so you should not use these Collection methods unless you really have to.

## BlockingQueue Implementations

Since **BlockingQueue** is an interface, you need to use one of its implementations to use it. Thejava.util.concurrent package has the following implementations of the **BlockingQueue** interface (in Java 6):

* [**ArrayBlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/arrayblockingqueue.html)
* [**DelayQueue**](http://tutorials.jenkov.com/java-util-concurrent/delayqueue.html)
* [**LinkedBlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/linkedblockingqueue.html)
* [**PriorityBlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/priorityblockingqueue.html)
* [**SynchronousQueue**](http://tutorials.jenkov.com/java-util-concurrent/synchronousqueue.html)

Click the links in the list to read more about each implementation. If a link cannot be clicked, that implementation has not yet been described. Check back again in the future, or check out the JavaDoc's for more detail.

## Java BlockingQueue Example

Here is a Java **BlockingQueue** example. The example uses the **ArrayBlockingQueue** implementation of the **BlockingQueue** interface.

First, the **BlockingQueueExample** class which starts a Producer and a Consumer in separate threads. The Producer inserts strings into a shared **BlockingQueue**, and the Consumer takes them out.

**package** com.test.blocking.queue.example;

**import** java.util.concurrent.ArrayBlockingQueue;

**import** java.util.concurrent.BlockingQueue;

**public** **class** BlockingQueueExample {

**public** **static** **void** main(String[] args) **throws** Exception {

BlockingQueue<String> queue = **new** ArrayBlockingQueue<String>(1024);

Producer producer = **new** Producer(queue);

Consumer consumer = **new** Consumer(queue);

**new** Thread(producer).start();

**new** Thread(consumer).start();

Thread.*sleep*(4000);

}

}

**package** com.test.blocking.queue.example;

**import** java.util.concurrent.BlockingQueue;

**public** **class** Consumer **implements** Runnable{

**protected** BlockingQueue<String> queue = **null**;

**public** Consumer(BlockingQueue<String> queue) {

**this**.queue = queue;

}

**public** **void** run() {

**try** {

System.*out*.println(queue.take());

System.*out*.println(queue.take());

System.*out*.println(queue.take());

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**package** com.test.blocking.queue.example;

**import** java.util.concurrent.BlockingQueue;

**public** **class** Producer **implements** Runnable{

**protected** BlockingQueue<String> queue = **null**;

**public** Producer(BlockingQueue<String> queue) {

**this**.queue = queue;

}

**public** **void** run() {

**try** {

queue.put("1");

Thread.*sleep*(1000);

queue.put("2");

Thread.*sleep*(1000);

queue.put("3");

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

**ArrayBlockingQueue** is a bounded, blocking queue that stores the elements internally in an array. That it is bounded means that it cannot store unlimited amounts of elements. There is an upper bound on the number of elements it can store at the same time. You set the upper bound at instantiation time, and after that it cannot be changed.

The **ArrayBlockingQueue** stores the elements internally in FIFO (First In, First Out) order. The head of the queue is the element which has been in queue the longest time, and the tail of the queue is the element which has been in the queue the shortest time.

Here is how to instantiate and use an ArrayBlockingQueue:

BlockingQueue queue = new ArrayBlockingQueue(1024);

queue.put("1");

Object object = queue.take();

Here is a **BlockingQueue** example that uses Java Generics. Notice how you can put and take String's instead of :

BlockingQueue<String> queue = new ArrayBlockingQueue<String>(1024);

queue.put("1");

String string = queue.take();

The **DelayQueue** blocks the elements internally until a certain delay has expired. The elements must implement the interface java.util.concurrent.Delayed. Here is how the interface looks:

public interface Delayed extends Comparable<Delayed< {

public long getDelay(TimeUnit timeUnit);

}

The value returned by the **getDelay**() method should be the delay remaining before this element can be released. If 0 or a negative value is returned, the delay will be considered expired, and the element released at the next take() etc. call on the DelayQueue.

The TimeUnit instance passed to the **getDelay**() method is an Enum that tells which time unit the delay should be returned in. The TimeUnit enum can take these values:

DAYS

HOURS

MINUTES

SECONDS

MILLISECONDS

MICROSECONDS

NANOSECONDS

The Delayed interface also extends the java.lang.Comparable interface, as you can see, which means that Delayed objects can be compared to each other. This is probably used internally in the **DelayQueue** to order the elements in the queue, so they are released ordered by their expiration time.

Here is an example of how to use the **DelayQueue**:

public class DelayQueueExample {

public static void main(String[] args) {

DelayQueue queue = new DelayQueue();

Delayed element1 = new DelayedElement();

queue.put(element1);

Delayed element2 = queue.take();

}

}

The DelayedElement is an implementation of the Delayed interface that I have created. It is not part of thejava.util.concurrent package. You will have to create your own implementation of the Delayed interface to use the DelayQueue class.

**SynchronousQueue.** It is an implementation of [BlockingQueue](http://javapapers.com/java/java-blockingqueue/). Among all [Java concurrent collections](http://javapapers.com/category/java/concurrent-util/), SynchronousQueue is different. Capacity of a synchrounous queue is always zero. It is because in SynchronousQueuean insert will wait for a remove operation by another thread and vice versa.

* put() call to a **SynchronousQueue** will not return until there is a corresponding take() call.
* peek is not possible with a **SynchronousQueue**
* As there is no element iteration is also not possible.
* Insert is not possible if there is a thread trying to remove it.
* **SynchronousQueue** should be imagined like a baton in a relay race.
* If there are more than one thread waiting for a  removal so that they can do insert then with fairness set to true, threads are granted access in FIFO order.
* **SynchronousQueue** is the default **BlockingQueue** used for theExecutors.newCachedThreadPool() methods.

**package** com.test.blocking.queue.example;

**import** java.util.concurrent.BlockingQueue;

**import** java.util.concurrent.SynchronousQueue;

**public** **class** SynchronousQueueExample {

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

**final** BlockingQueue<String> synchronousQueue = **new** SynchronousQueue<String>();

SynchronousQueueProducer queueProducer = **new** SynchronousQueueProducer(

synchronousQueue);

**new** Thread(queueProducer).start();

SynchronousQueueConsumer queueConsumer1 = **new** SynchronousQueueConsumer(

synchronousQueue);

**new** Thread(queueConsumer1).start();

SynchronousQueueConsumer queueConsumer2 = **new** SynchronousQueueConsumer(

synchronousQueue);

**new** Thread(queueConsumer2).start();

}

}

**package** com.test.blocking.queue.example;

**import** java.util.Random;

**import** java.util.UUID;

**import** java.util.concurrent.BlockingQueue;

**public** **class** SynchronousQueueProducer **implements** Runnable {

**protected** BlockingQueue<String> blockingQueue;

**final** Random random = **new** Random();

**public** SynchronousQueueProducer(BlockingQueue<String> queue) {

**this**.blockingQueue = queue;

}

@Override

**public** **void** run() {

**while** (**true**) {

**try** {

String data = UUID.*randomUUID*().toString();

System.*out*.println("Put: " + data);

blockingQueue.put(data);

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**package** com.test.blocking.queue.example;

**import** java.util.concurrent.BlockingQueue;

**public** **class** SynchronousQueueConsumer **implements** Runnable {

**protected** BlockingQueue<String> blockingQueue;

**public** SynchronousQueueConsumer(BlockingQueue<String> queue) {

**this**.blockingQueue = queue;

}

@Override

**public** **void** run() {

**while** (**true**) {

**try** {

String data = blockingQueue.take();

System.*out*.println(Thread.*currentThread*().getName()

+ " take(): " + data);

Thread.*sleep*(2000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

The **LinkedBlockingQueue** class implements the [**BlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) interface. Read the [BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) text for more information about the interface.

The **LinkedBlockingQueue** keeps the elements internally in a linked structure (linked nodes). This linked structure can optionally have an upper bound if desired. If no upper bound is specified,**Integer.MAX\_VALUE** is used as the upper bound.

The **LinkedBlockingQueue** stores the elements internally in FIFO (First In, First Out) order. The head of the queue is the element which has been in queue the longest time, and the tail of the queue is the element which has been in the queue the shortest time.

Here is how to instantiate and use a **LinkedBlockingQueue**:

BlockingQueue<String> unbounded = new LinkedBlockingQueue<String>();

BlockingQueue<String> bounded = new LinkedBlockingQueue<String>(1024);

bounded.put("Value");

String value = bounded.take();

The **PriorityBlockingQueue** class implements the [**BlockingQueue**](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) interface. Read the [BlockingQueue](http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html) text for more information about the interface.

The **PriorityBlockingQueue** is an unbounded concurrent queue. It uses the same ordering rules as the java.util.**PriorityQueue** class. You cannot insert null into this queue.

All elements inserted into the **PriorityBlockingQueue** must implement the **java.lang.Comparable** interface. The elements thus order themselves according to whatever priority you decide in your Comparable implementation.

Notice that the **PriorityBlockingQueue** does not enforce any specific behavior for elements that have equal priority (compare() == 0).

Also notice, that in case you obtain an Iterator from a **PriorityBlockingQueue**, the Iterator does not guarantee to iterate the elements in priority order.

Here is an example of how to use the **PriorityBlockingQueue**:

BlockingQueue queue = new PriorityBlockingQueue();

//String implements java.lang.Comparable

queue.put("Value");

String value = queue.take();

**package** com.test.blocking.queue.example;

**import** java.util.concurrent.BlockingQueue;

**import** java.util.concurrent.LinkedBlockingQueue;

**public** **class** PriorityBlockingQueueExample {

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

**final** BlockingQueue<String> priorityBlockingQueue = **new** LinkedBlockingQueue<String>();

PriorityBlockingQueueProducer queueProducer = **new** PriorityBlockingQueueProducer(

priorityBlockingQueue);

**new** Thread(queueProducer).start();

PriorityBlockingQueueConsumer queueConsumer1 = **new** PriorityBlockingQueueConsumer(

priorityBlockingQueue);

**new** Thread(queueConsumer1).start();

PriorityBlockingQueueConsumer queueConsumer2 = **new** PriorityBlockingQueueConsumer(

priorityBlockingQueue);

**new** Thread(queueConsumer2).start();

}

}

**package** com.test.blocking.queue.example;

**import** java.util.Random;

**import** java.util.UUID;

**import** java.util.concurrent.BlockingQueue;

**public** **class** PriorityBlockingQueueProducer **implements** Runnable {

**protected** BlockingQueue<String> blockingQueue;

**final** Random random = **new** Random();

**public** PriorityBlockingQueueProducer(BlockingQueue<String> queue) {

**this**.blockingQueue = queue;

}

@Override

**public** **void** run() {

**while** (**true**) {

**try** {

String data = UUID.*randomUUID*().toString();

System.*out*.println("Put: " + data);

blockingQueue.put(data);

Thread.*sleep*(500);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

**package** com.test.blocking.queue.example;

**import** java.util.concurrent.BlockingQueue;

**public** **class** PriorityBlockingQueueConsumer **implements** Runnable {

**protected** BlockingQueue<String> blockingQueue;

**public** PriorityBlockingQueueConsumer(BlockingQueue<String> queue) {

**this**.blockingQueue = queue;

}

@Override

**public** **void** run() {

**while** (**true**) {

**try** {

String data = blockingQueue.take();

System.*out*.println(Thread.*currentThread*().getName()

+ " take(): " + data);

Thread.*sleep*(1000);

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

}

}

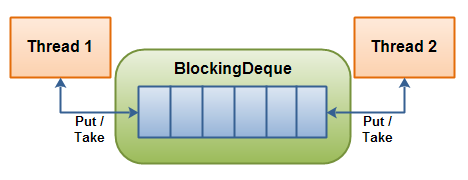
The **BlockingDeque** interface in the java.util.concurrent class represents a deque which is thread safe to put into, and take instances from. In this text I will show you how to use this **BlockingDeque**.

The **BlockingDeque** class is a **Deque** which blocks threads tring to insert or remove elements from the deque, in case it is either not possible to insert or remove elements from the deque.

A deque is short for "Double Ended Queue". Thus, a deque is a queue which you can insert and take elements from, from both ends.

BlockingDeque Usage

A **BlockingDeque** could be used if threads are both producing and consuming elements of the same queue. It could also just be used if the producing thread needs to insert at both ends of the queue, and the consuming thread needs to remove from both ends of the queue. Here is an illustration of that:



**A BlockingDeque - threads can put and take from both ends of the deque.**

A thread will produce elements and insert them into either end of the queue. If the deque is currently full, the inserting thread will be blocked until a removing thread takes an element out of the deque. If the deque is currently empty, a removing thread will be blocked until an inserting thread inserts an element into the deque.

**BlockingDeque methods:**

A **BlockingDeque** has 4 different sets of methods for inserting, removing and examining the elements in the deque. Each set of methods behaves differently in case the requested operation cannot be carried out immediately. Here is a table of the methods:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws Exception** | **Special Value** | **Blocks** | **Times Out** |
| Insert | addFirst(o) | offerFirst(o) | putFirst(o) | offerFirst(o, timeout, timeunit) |
| Remove | removeFirst(o) | pollFirst(o) | takeFirst(o) | pollFirst(timeout, timeunit) |
| Examine | getFirst(o) | peekFirst(o) |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Throws Exception** | **Special Value** | **Blocks** | **Times Out** |
| Insert | addLast(o) | offerLast(o) | putLast(o) | offerLast(o, timeout, timeunit) |
| Remove | removeLast(o) | pollLast(o) | takeLast(o) | pollLast(timeout, timeunit) |
| Examine | getLast(o) | peekLast(o) |  |  |

The 4 different sets of behaviour means this:

Throws Exception:   
If the attempted operation is not possible immediately, an exception is thrown.

Special Value:   
If the attempted operation is not possible immediately, a special value is returned (often true / false).

Blocks:   
If the attempted operation is not possible immedidately, the method call blocks until it is.

Times Out:   
If the attempted operation is not possible immedidately, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).

**BlockingDeque** Extends **BlockingQueue**

The **BlockingDeque** interface extends the **BlockingQueue** interface. That means that you can use a **BlockingDeque** as a **BlockingQueue**. If you do so, the various inserting methods will add the elements to the end of the deque, and the removing methods will remove the elements from the beginning of the deque.

The inserting and removing methods of the BlockingQueue interface, that is.

Here is a table of what the methods of the **BlockingQueue** does in a **BlockingDeque** implementation:

|  |  |
| --- | --- |
| BlockingQueue | BlockingDeque |
| add() | addLast() |
| offer() x 2 | offerLast() x 2 |
| put() | putLast() |
|  |  |
| remove() | removeFirst() |
| poll() x 2 | pollFirst() |
| take() | takeFirst() |
|  |  |
| element() | getFirst() |
| peek() | peekFirst() |

**BlockingDeque** Implementations

Since BlockingDeque is an interface, you need to use one of its many implementations to use it. Thejava.util.concurrent package has the following implementations of the BlockingDeque interface:

[**LinkedBlockingDeque**](http://tutorials.jenkov.com/java-util-concurrent/linkedblockingdeque.html)

**BlockingDeque** Code Example

Here is a small code example of how to use the **BlockingDeque** methods:

BlockingDeque<String> deque = new LinkedBlockingDeque<String>();

deque.addFirst("1");

deque.addLast("2");

String two = deque.takeLast();

String one = deque.takeFirst();

The **LinkedBlockingDeque** class implements the [**BlockingDeque**](http://tutorials.jenkov.com/java-util-concurrent/blockingdeque.html) interface. Read the [BlockingDeque](http://tutorials.jenkov.com/java-util-concurrent/blockingdeque.html) text for more information about the interface.

The word **Deque** comes from the term "Double Ended Queue". A **Deque** is thus a queue where you can insert and remove elements from both ends of the queue.

The **LinkedBlockingDeque** is a **Deque** which will block if a thread attempts to take elements out of it while it is empty, regardless of what end the thread is attempting to take elements from.

Here is how to instantiate and use a **LinkedBlockingDeque**:

BlockingDeque<String> deque = new LinkedBlockingDeque<String>();

deque.addFirst("1");

deque.addLast("2");

String two = deque.takeLast();

String one = deque.takeFirst();

A **java.util.concurrent.locks.Lock** is a thread synchronization mechanism just like synchronized blocks. A **Lock** is, however, more flexible and more sophisticated than a synchronized block.

By the way, in my [Java Concurrency tutorial](http://tutorials.jenkov.com/java-concurrency/index.html) I have described how to implement your own locks, in case you are interested (or need it). See my text on [Locks](http://tutorials.jenkov.com/java-concurrency/locks.html) for more details.

Java Lock Example

Since Lock is an interface, you need to use one of its implementations to use a Lock in your applications. Here is a simple usage example:

Lock lock = new ReentrantLock();

lock.lock();

//critical section

lock.unlock();

First a Lock is created. Then it's **lock()** method is called. Now the Lock instance is locked. Any other thread calling **lock()** will be blocked until the thread that locked the lock calls unlock(). Finally **unlock()** is called, and the **Lock** is now unlocked so other threads can lock it.

## Java Lock Implementations

The java.util.concurrent.locks package has the following implementations of the Lock interface:

* ReentrantLock

## Main Differences between Locks and Synchronized Blocks

The main differences between a Lock and a synchronized block are:

* A synchronized block makes no guarantees about the sequence in which threads waiting to entering it are granted access.
* You cannot pass any parameters to the entry of a synchronized block. Thus, having a timeout trying to get access to a synchronized block is not possible.
* The synchronized block must be fully contained within a single method. A Lock can have it's calls tolock() and unlock() in separate methods.

## Lock Methods

The Lock interface has the following primary methods:

* **lock()**
* **lockInterruptibly()**
* **tryLock()**
* **tryLock(long timeout, TimeUnit timeUnit)**
* **unlock()**

The **lock()** method locks the Lock instance if possible. If the Lock instance is already locked, the thread calling lock() is blocked until the Lock is unlocked.

The **lockInterruptibly()** method locks the Lock unless the thread calling the method has been interrupted. Additionally, if a thread is blocked waiting to lock the Lock via this method, and it is interrupted, it exits this method calls.

The **tryLock()** method attempts to lock the Lock instance immediately. It returns true if the locking succeeds, false if Lock is already locked. This method never blocks.

The **tryLock(long timeout, TimeUnit timeUnit)** works like **the tryLock()** method, except it waits up the given timeout before giving up trying to lock the Lock.

The **unlock()** method unlocks the Lock instance. Typically, a Lock implementation will only allow the thread that has locked the Lock to call this method. Other threads calling this method may result in an unchecked exception (RuntimeException).