**java.util.concurrent.CopyOnWriteArraySet<E>**

**public class CopyOnWriteArraySet<E> extends AbstractSet<E> implements Serializable**

A Set that uses an internal CopyOnWriteArrayList for all of its operations. Thus, it shares the same basic properties:

* It is best suited for applications in which set sizes generally stay small, read-only operations vastly outnumber mutative operations, and you need to prevent interference among threads during traversal.
* It is thread-safe.
* Mutative operations (add, set, remove, etc.) are expensive since they usually entail copying the entire underlying array.
* Iterators do not support the mutative remove operation.
* Traversal via iterators is fast and cannot encounter interference from other threads. Iterators rely on unchanging snapshots of the array at the time the iterators were constructed.

**CopyOnWriteArraySet share some properties of Set and also has its own properties:**

* The internal implementation of CopyOnWriteArraySet is CopyOnWriteArrayList only.
* Insertion order is preserved and duplicates are allowed.
* Multiple Threads are able to perform update operation simultaneously but for every update operation a **separate cloned copy** is created. As for every update a new cloned copy will be created which is costly. Hence if multiple update operation are required then it is not recommended to use CopyOnWriteArraySet.
* While one thread iterating the Set, other threads can perform updation, here we wont get any runtime exception like ConcurrentModificationException.
* Iterator of CopyOnWriteArraySet class can perform only read only and wont perform deletion, otherwise we will get Run-time exception UnsupportedOperationException.

**The important things to learn about Java CopyOnWriteArraySet class are:**

* As normal set data structure, it does not allow duplicates.
* CopyOnWriteArraySet class implement Serializable interface and extends AbstractSet class.
* Using CopyOnWriteArraySet is costly for update operations, bacause each mutation creates a cloned copy of underlying array and add/update element to it.
* It is thread-safe version of HashSet. Each thread accessing the set sees its own version of snapshot of backing array created while initializing the iterator for this set.
* Because it gets snapshot of underlying array while creating iterator, it **does not throw ConcurrentModificationException**.
* Mutation operations on iterators are not supported. These methods throw UnsupportedOperationException.
* CopyOnWriteArraySet is a concurrent replacement for a **synchronized Set** and offers better concurrency when iterations outnumber mutations.
* It allows duplicate elements and heterogeneous Objects (use generics to get compile time errors).
* Because it creates a new copy of underlying array everytime iterator is created, **performance is slower** than HashSet.

## **Java CopyOnWriteArraySet Usecases**

Use CopyOnWriteArraySet in applications in which set sizes generally stay small, read-only operations vastly outnumber mutative operations, and you need to prevent interference among threads during traversal.

CopyOnWriteArraySet helps in minimizing programmer controlled synchronization steps and move control to inbuilt, well tested APIs.

## **Java CopyOnWriteArraySet Performance**

Due to added step of creating a new backing array everytime the set is updated, it performs worse than HashSet.  
There is no performance overhead on read operations and both classes perform same.

## **CopyOnWriteArraySet Methods**

* **boolean add(object o)** : Adds the specified element to this set if it is not already present.
* **boolean addAll(collection c)** : Adds all of the elements in the specified collection to this set if they’re not already present.
* **void clear()** : Removes all of the elements from this set.
* **boolean contains(Object o)** : Returns true if this set contains the specified element.
* **boolean isEmpty()** : Returns true if this set contains no elements.
* **Iterator iterator()** : Returns an iterator over the elements contained in this set in the order in which these elements were added.
* **boolean remove(Object o)** : Removes the specified element from this set if it is present.
* **int size()** : Returns the number of elements in this set.

**java.util.Collections.synchronizedSet() Method**

The **synchronizedSet()** method of **java.util.Collections** class is used to return a synchronized (thread-safe) set backed by the specified set. In order to guarantee serial access, it is critical that all access to the backing set is accomplished through the returned set.

**Syntax:**

public static <T> Set<T> synchronizedSet(Set<T> s)

**Parameters:** This method takes the **set** as a parameter to be “wrapped” in a synchronized set.

**Return Value:** This method returns a **synchronized view** of the specified set.

## **CopyOnWriteArrayList v/s SynchronizedSet :**

* **COWAL –>** CopyOnWriteArrayList
* **COWAS –>** CopyOnWriteArraySet

|  |  |
| --- | --- |
| **CopyOnWriteArraySet** | **SynchronizedSet** |
| **CopyOnWriteArrayList** is newly introduced **thread-safe** class (i.e.; synchronized) | This is **thread-safe** version of Set i.e.; any Set implemented classes like HashSet or TreeSet can be converted into synchronized Set (thread-safe) |
| **Multiple threads** are allowed to operate on **CopyOnWriteArraySet**, as it works on separate **cloned copy** for **update/modify** operations | Only **one thread** is allowed to operate on synchronized set, by **locking over complete list** object |
| While one thread **iterating** **CopyOnWriteArraySet** object, other threads happily can **modify**, as it works on separate cloned copy  And it **never** throws **ConcurrentModificationException** | While one thread **iterating** synchronized set object, if any other threads tries to **modify** the same Set object then **ConcurrentModificationException** is thrown |
| That’s it is **fail-safe iterator** | That’s it is **fail-fast iterator** |
| There is **no such restriction** while iterating on **CopyOnWriteArraySet**;  We can safely iterate **outside synchronized block** | While **iterating synchronized Set**, make sure to **iterate inside synchronized block**;  Otherwise we may face **non-deterministic behavior** |
| **Iterator of CopyOnWriteArraySet** can perform **read operation** safely; while iterating through **COWAS** items  But as soon as, **remove** operation is performed, compiler throws **UnsupportedOperationException** | **Iterator** of **Set** can perform both **read** and **remove** operations; while iterating through Set elements |
| This is introduced in **Java 1.5** version | This is introduced in original collection framework in **Java 1.2** version |

### **When to use SynchronizedSet ?**

* This is generally used to **convert** set object into **thread-safe** set object
* But only one thread is allowed to operate on set object, as **lock** is acquired **over complete set** object
* So, **performance degrades** comparatively in a multi-threaded environment
* So, **use this only** when it is required to convert into thread-safe version of Set object
* Otherwise, mature and latest **CopyOnWriteArraySet** can be used efficiently

### **When to use CopyOnWriteArraySet ?**

* This is the **best suit to store unique elements as per insertion order** in a multi-threaded environment
* Where there are **more number of read** operation and **very less update**/modify operation
* Because for every update/modify operations, a **new separate cloned copy** is created
* And there is **overhead on JVM** to allocate **memory** & **merging** cloned copy with original copy
* The **advantage** of using CopyOnWriteArraySet over Set is that, it doesn’t throws **ConcurrentModificationException**when **multiple threads** perform operation simultaneously

### **CopyOnWriteArraySet v/s SynchronizedSet :**

* there is always a catch between **performance** and **thread-safety**
* choose wisely for your requirement

# **CopyOnWriteArraySet v/s HashSet :**

* **COWAL** –> CopyOnWriteArrayList
* **COWAS** –> CopyOnWriteArraySet

|  |  |
| --- | --- |
| **CopyOnWriteArraySet** | **HashSet** |
| CopyOnWriteArraySet is **synchronized** and it is **internally implemented** using **COWAL** | HashSet is **not** synchronized |
| For every **update** operation, a new separate cloned copy is created and there is **memory & merging overhead** for **JVM**  Hence**, performance** is relatively **low** when comparing with HashSet | In multi-threaded environment, **HashSet** is **faster** than **CopyOnWriteArraySet** as **multiple threads can operate**  Hence, **performance** is **high** as there is **no need to acquire lock** |
| While one thread **iterating CopyOnWriteArraySet** items, other threads happily can **modify**, as it works on separate cloned copy  And it **never** throws **ConcurrentModificationException** | While one thread **iterating** HashSet items, if any other thread tries to **modify** same HashSet object then **ConcurrentModificationException** is thrown |
| That’s it is **fail-safe iterator** | That’s it is **fail-fast iterator** |
| **Iterator of CopyOnWriteArraySet** can perform **read operation** safely; while iterating through **COWAS** items  But as soon as, **remove** operation is performed, compiler throws **UnsupportedOperationException** | **Iterator** of **HashSet** can perform both **read** and **remove** operations; while iterating through HashSet elements |
| Present in **java.util.concurrent** package and qualified class name is **java.util.concurrent** **.CopyOnWriteArraySet** | Present in **java.util** package and qualified class name is **java.util.HashSet** |
| This is introduced in **Java 1.5** version | This is introduced in original collection framework in **Java 1.2** version |

### **When to use HashSet ?**

* HashSet stores **unique elements** using **hashing** technique
* So, **search operation** is faster
* So, if business requirement is to store **unique elements** for faster **search operation** or **more number of search operation** without concerning **insertion order**
* Then, HashSet is the very apt choice

### **When to use CopyOnWriteArraySet ?**

* This is the **best suit to store unique elements as per insertion order** in a multi-threaded environment
* Where there are **more number of read** operation and **very less update**/modify operation
* Because for every update/modify operations, a **new separate cloned copy** is created
* And there is **overhead on JVM** to allocate **memory** & **merging** cloned copy with original copy
* The **advantage** of using CopyOnWriteArraySet over HashSet is that, it doesn’t throws **ConcurrentModificationException** when **multiple threads** performs operation simultaneously

### **CopyOnWriteArraySet v/s HashSet :**

* there is always a catch between **performance** and **thread-safety**
* choose wisely for your requirement

**java.util.concurrent.ConcurrentHashMap.KeySetView<K,V>**

**public static class ConcurrentHashMap.KeySetView<K,V> extends Object implements Set<K>, Serializable**

Utility classes commonly useful in concurrent programming.

A view of a ConcurrentHashMap as a Set of keys, in which additions may optionally be enabled by mapping to a common value. This class cannot be directly instantiated. See keySet(), keySet(V), newKeySet(), newKeySet(int).

ConcurrentHashMap.newKeySet() should be somewhat more efficient as removes a single level of indirection. Collections.newSetFromMap(map) is mostly based on redirecting the operations to the map.keySet(), but ConcurrentHashMap.newKeySet() is very close to map.keySet() itself (just with additions support).

### *Uses of* [*ConcurrentHashMap.KeySetView*](https://jutge.org/doc/java/docs/api/java/util/concurrent/ConcurrentHashMap.KeySetView.html) *in* [*java.util.concurrent*](https://jutge.org/doc/java/docs/api/java/util/concurrent/package-summary.html)

Methods in [java.util.concurrent](https://jutge.org/doc/java/docs/api/java/util/concurrent/package-summary.html) that return [ConcurrentHashMap.KeySetView](https://jutge.org/doc/java/docs/api/java/util/concurrent/ConcurrentHashMap.KeySetView.html)

|  |  |  |
| --- | --- | --- |
| **Modifier and Type** | **Method** | **Description** |
| [**ConcurrentHashMap.KeySetView**](https://jutge.org/doc/java/docs/api/java/util/concurrent/ConcurrentHashMap.KeySetView.html)**<**[**K**](https://jutge.org/doc/java/docs/api/java/util/concurrent/ConcurrentHashMap.html)**,**[**V**](https://jutge.org/doc/java/docs/api/java/util/concurrent/ConcurrentHashMap.html)**>** | **ConcurrentHashMap.**[**keySet**](https://jutge.org/doc/java/docs/api/java/util/concurrent/ConcurrentHashMap.html#keySet--) **()** | Returns a [**Set**](https://jutge.org/doc/java/docs/api/java/util/Set.html) **view of the keys contained in this map.** |
| [**ConcurrentHashMap.KeySetView**](https://jutge.org/doc/java/docs/api/java/util/concurrent/ConcurrentHashMap.KeySetView.html)**<**[**K**](https://jutge.org/doc/java/docs/api/java/util/concurrent/ConcurrentHashMap.html)**,**[**V**](https://jutge.org/doc/java/docs/api/java/util/concurrent/ConcurrentHashMap.html)**>** | **ConcurrentHashMap.**[**keySet**](https://jutge.org/doc/java/docs/api/java/util/concurrent/ConcurrentHashMap.html#keySet-V-) **(**[**V**](https://jutge.org/doc/java/docs/api/java/util/concurrent/ConcurrentHashMap.html) **mappedValue)** | Returns a [**Set**](https://jutge.org/doc/java/docs/api/java/util/Set.html) **view of the keys in this map, using the given common mapped value for any additions (i.e.,** [**Collection.add(E)**](https://jutge.org/doc/java/docs/api/java/util/Collection.html#add-E-) **and** [**Collection.addAll(Collection)**](https://jutge.org/doc/java/docs/api/java/util/Collection.html#addAll-java.util.Collection-)**).** |
| static <K> [**ConcurrentHashMap.KeySetView**](https://jutge.org/doc/java/docs/api/java/util/concurrent/ConcurrentHashMap.KeySetView.html)**<K,**[**Boolean**](https://jutge.org/doc/java/docs/api/java/lang/Boolean.html)**>** | **ConcurrentHashMap.**[**newKeySet**](https://jutge.org/doc/java/docs/api/java/util/concurrent/ConcurrentHashMap.html#newKeySet--) **()** | Creates a new [**Set**](https://jutge.org/doc/java/docs/api/java/util/Set.html) **backed by a ConcurrentHashMap from the given type to** Boolean.TRUE. |
| static <K> [**ConcurrentHashMap.KeySetView**](https://jutge.org/doc/java/docs/api/java/util/concurrent/ConcurrentHashMap.KeySetView.html)**<K,**[**Boolean**](https://jutge.org/doc/java/docs/api/java/lang/Boolean.html)**>** | **ConcurrentHashMap.**[**newKeySet**](https://jutge.org/doc/java/docs/api/java/util/concurrent/ConcurrentHashMap.html#newKeySet-int-) **(int initialCapacity)** | Creates a new [**Set**](https://jutge.org/doc/java/docs/api/java/util/Set.html) **backed by a ConcurrentHashMap from the given type to** Boolean.TRUE. |

**java.util.concurrent.ConcurrentSkipListSet<E>**

**public class ConcurrentSkipListSet<E>extends AbstractSet<E>implements NavigableSet<E>, Cloneable, Serializable**

* A scalable concurrent NavigableSet implementation based on a ConcurrentSkipListMap.
* The elements of the set are kept sorted according to their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.
* This implementation provides expected average log(n) time cost for the contains, add, and remove operations and their variants. Insertion, removal, and access operations safely execute concurrently by multiple threads.
* Iterators and spliterators are weakly consistent.
* Ascending ordered views and their iterators are faster than descending ones.
* Beware that, unlike in most collections, the size method is not a constant-time operation. Because of the asynchronous nature of these sets, determining the current number of elements requires a traversal of the elements, and so may report inaccurate results if this collection is modified during traversal. Additionally, the bulk operations addAll, removeAll, retainAll, containsAll, equals, and toArray are not guaranteed to be performed atomically. For example, an iterator operating concurrently with an addAll operation might view only some of the added elements.
* This class and its iterators implement all of the optional methods of the Set and Iterator interfaces. Like most other concurrent collection implementations, this class does not permit the use of null elements, because null arguments and return values cannot be reliably distinguished from the absence of elements.
* The ConcurrentSkipListSet class in Java is a part of the Java Collection Framework and implements the Collection interface and the AbstractSet class. It provides a scalable and concurrent version of NavigableSet in Java. The implementation of ConcurrentSkipListSet is based on ConcurrentSkipListMap. The elements in ConcurrentSkipListSet are sorted by default in their natural ordering.

**Methods in Java ConcurrentSkipListSet:**

1. [**add(E e)**](https://www.geeksforgeeks.org/concurrentskiplistset-add-method-in-java/)**: This method adds the specified element to this set if it is not already present.**
2. [**ceiling(E e)**](https://www.geeksforgeeks.org/concurrentskiplistset-ceiling-method-in-java/)**: This method returns the least element in this set greater than or equal to the given element, or null if there is no such element.**
3. [**clear()**](https://www.geeksforgeeks.org/concurrentskiplistset-clear-method-in-java/)**: This method removes all of the elements from this set.**
4. [**clone()**](https://www.geeksforgeeks.org/concurrentskiplistset-clone-method-in-java/)**: This method returns a shallow copy of this ConcurrentSkipListSet instance.**
5. **comparator():** This method returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements.
6. [**contains(Object o)**](https://www.geeksforgeeks.org/concurrentskiplistset-contains-method-in-java/)**: This method returns true if this set contains the specified element.**
7. **descendingIterator()**: This method returns an iterator over the elements in this set in descending order.
8. [**descendingSet()**](https://www.geeksforgeeks.org/concurrentskiplistset-descendingset-method-in-java/)**: This method returns a reverse order view of the elements contained in this set.**
9. [**equals(Object o)**](https://www.geeksforgeeks.org/concurrentskiplistset-equals-method-in-java/)**: This method compares the specified object with this set for equality.**
10. [**first()**](https://www.geeksforgeeks.org/concurrentskiplistset-first-method-in-java/)**: This method returns the first (lowest) element currently in this set.**
11. **floor(E e)**: This method returns the greatest element in this set less than or equal to the given element, or null if there is no such element.
12. **headSet(E toElement)**: This method returns a view of the portion of this set whose elements are strictly less than toElement.
13. **headSet(E toElement, boolean inclusive)**: This method returns a view of the portion of this set whose elements are less than (or equal to, if inclusive is true) toElement.
14. **higher(E e)**: This method returns the least element in this set strictly greater than the given element, or null if there is no such element.
15. [**isEmpty()**](https://www.geeksforgeeks.org/concurrentskiplistset-isempty-method-in-java/)**: This method returns true if this set contains no elements.**
16. [**iterator()**](https://www.geeksforgeeks.org/concurrentskiplistset-iterator-method-in-java/)**: This method returns an iterator over the elements in this set in ascending order.**
17. **last():** This method returns the last (highest) element currently in this set.
18. **lower(E e)**: This method returns the greatest element in this set strictly less than the given element, or null if there is no such element.
19. **pollFirst()**: This method retrieves and removes the first (lowest) element, or returns null if this set is empty.
20. **pollLast()**: This method retrieves and removes the last (highest) element, or returns null if this set is empty.
21. **remove(Object o)**: This method removes the specified element from this set if it is present.
22. [**removeAll(Collection<E> c)**](https://www.geeksforgeeks.org/concurrentskiplistset-removeall-method-in-java/)**: This method removes from this set all of its elements that are contained in the specified collection.**
23. [**size()**](https://www.geeksforgeeks.org/concurrentskiplistset-size-method-in-java/)**: This method returns the number of elements in this set.**
24. **spliterator()**: This method returns a Spliterator over the elements in this set.
25. **subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive)**: This method returns a view of the portion of this set whose elements range from fromElement to toElement.
26. **subSet(E fromElement, E toElement)**: This method returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
27. **tailSet(E fromElement):** This method returns a view of the portion of this set whose elements are greater than or equal to fromElement.
28. **tailSet(E fromElement, boolean inclusive)**: This method returns a view of the portion of this set whose elements are greater than (or equal to, if inclusive is true) fromElement.

**When is a ConcurrentSkipListSet useful?**

**Answer: -**

* ConcurrentSkipListSet and ConcurrentSkipListMap are useful when you need a sorted container that will be accessed by multiple threads. These are essentially the equivalents of TreeMap and TreeSet for concurrent code.
* The implementation for JDK 6 is based on High Performance Dynamic Lock-Free Hash Tables and List-Based Sets by Maged Michael at IBM, which shows that you can implement a lot of operations on skip lists atomically using compare and swap (CAS) operations. These are lock-free, so you don't have to worry about the overhead of synchronized (for most operations) when you use these classes.
* There's currently no Red-Black tree based concurrent Map/Set implementation in Java. I looked through the literature a bit and found a couple papers that showed concurrent RB trees outperforming skip lists, but a lot of these tests were done with transactional memory, which isn't supported in hardware on any major architectures at the moment.
* I'm assuming the JDK guys went with a skip list here because the implementation was well-known and because making it lock-free was simple and portable (using CAS). If anyone cares to clarify, please do. I'm curious.

**Similarity between TreeSet and ConcurrentSkipListSet:**

* Both TreeSet and ConcurrentSkipListSet will throw NullPointerException if thre is any attempt to add NULL elements.
* Both TreeSet and ConcurrentSkipListSet extends AbstractSet and implements interfaces Set,SortedSet and NavigableSet.
* Both TreeSet and ConcurrentSkipListSet doesnot allow duplicate elements.
* Both are sorted by natural order or according to the comparator provided.

**Differences between TreeSet and ConcurrentSkipListSet:**

* TreeSet belongs to java.util package and ConcurrentSkipListSet belongs to util.concurrent**.**
* TreeSet is not synchronized so multiple threads can access same TreeSet object which is not possible in case of ConcurrentSkipListSet as ConcurrentSkipListSet is synchronized.
* Iterator of TreeSet is FailFast which means that iterator will throw ConcurrentModificationException if there is any structural modification made to TreeSet during iteration whereas iterator of ConcurrentSkipListSet is FailSafe.
* Operation in TreeSet are faster as methods are not synchronized whereas operations on ConcurrentSkipListSet are slow because of synchronized method.
* TreeSet was introduced in JDK2.0 whereas ConcurrentSkipListSet is introduced in JDK6.0

**Difference between ConcurrentSkipListSet and SynchronizedSet**

**Answer: -** There are two primary methods to implement a synchronized set in Java with the help of ConcurrentSkipListSet class or Collections.synchronizedSet() method. This post provides an overview of some of the major dierences between these two.

* **Concurrency: -**The ConcurrentSkipListSet class allows safe execution of Insertion, removal, and access operations on set concurrently by multiple threads. It should be preferred over other implementations of the Set interface when concurrent modication of set by multiple threads is required. Collections.SynchronizedSet() locks the entire set which blocks parallel access for multiple threads. In other words, only one thread can access the set at a time which degrades the performance. Collections.SynchronizedSet() should be avoided unless data consistency is important.
* **Null values: -**The ConcurrentSkipListSet does not permit use of null values, whereas the behavior of Collections.SynchronizedSet() method depends on the backing set. For instance, if HashSet object is passed to SynchronizedSet(), null values are allowed and if a TreeSet object is passed, null values are not allowed.
* **Fail-safe or Fail-fast behavior: -**The ConcurrentSkipListSet‘s iterator is a weakly consistent which never throws ConcurrentModificationException even if the set is modied after the construction of the iterator. Note that its implementation makes no guarantees to reect any modications subsequent to the construction. On the other hand, Collections.SynchronizedSet() does not guarantee fail-safe iterator on concurrent modication (one thread is updating the set and another thread is traversing the set using an iterator). The fail-safe or fail-fast behavior totally depends on the backing set.
* **Iteration Order: -**ConcurrentSkipListSet does not preserve the insertion order of elements in the set and the elements are kept sorted according to their natural ordering, or by a provided Comparator. Collections.SynchronizedSet() is backed by the specied set and retains the insertion order of the set. If HashSet is passed to it, the order of the set is undened and if a TreeSet is used, the order is same as natural ordering of its elements, or that of a provided Comparator.

**java.util.EnumSet<E extends Enum<E>>**

**public abstract class EnumSet<E extends Enum<E>> extends AbstractSet<E> implements Cloneable, Serializable**

A specialized 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.

The iterator returned by the iterator method traverses the elements in their natural order (the order in which the enum constants are declared). The returned iterator is weakly consistent: it will never throw ConcurrentModificationException and it may or may not show the effects of any modifications to the set that occur while the iteration is in progress.

Null elements are not permitted. Attempts to insert a null element will throw NullPointerException. Attempts to test for the presence of a null element or to remove one will, however, function properly.

Like most collection implementations, EnumSet is not synchronized. If multiple threads access an enum set concurrently, and at least one of the threads modifies the set, it should be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the enum set. If no such object exists, the set should be "wrapped" using the Collections.synchronizedSet(java.util.Set<T>) method. This is best done at creation time, to prevent accidental unsynchronized access:

Set<MyEnum> s = Collections.synchronizedSet(EnumSet.noneOf(MyEnum.class));

Implementation note: All basic operations execute in constant time. They are likely (though not guaranteed) to be much faster than their HashSet counterparts. Even bulk operations execute in constant time if their argument is also an enum set.

EnumSet is one of the specialized implementation of [Set interface](http://quiz.geeksforgeeks.org/set-in-java/) for use with the [enumeration type](https://www.geeksforgeeks.org/enum-in-java/). Few important features of EnumSet are as follows:

* It extends AbstractSet and implements [Set Interface](http://quiz.geeksforgeeks.org/set-in-java/) in Java.
* EnumSet class is a member of the [Java Collections Framework](https://www.geeksforgeeks.org/collections-in-java-2/) & is not synchronized.
* It’s a high performance set implementation, much faster than [HashSet](http://quiz.geeksforgeeks.org/hashset-in-java/).
* All of the elements in an enum set must come from a single [enumeration type](https://www.geeksforgeeks.org/enum-in-java/) that is specified when the set is created either explicitly or implicitly.

**Methods in EnumSet:**

* [EnumSet of(E e1):](https://www.geeksforgeeks.org/enumset-of-method-in-java/) Creates an enum set initially containing the specified elements.
* [EnumSet of(E e11, E el2):](https://www.geeksforgeeks.org/enumset-of-method-in-java/) Creates an enum set initially containing the specified elements.
* [EnumSet of(E e11, E el2, Eel3):](https://www.geeksforgeeks.org/enumset-of-method-in-java/) Creates an enum set initially containing the specified elements.
* [EnumSet of(E e11, E el2, Eel3,E el4….):](https://www.geeksforgeeks.org/enumset-of-method-in-java/) Creates an enum set initially containing the specified elements.
* [EnumSet of(E e1, E rest…):](https://www.geeksforgeeks.org/enumset-of-method-in-java/) Creates an enum set initially containing the specified elements.
* [EnumSet complementOf(EnumSet s):](https://www.geeksforgeeks.org/enumset-complementof-method-in-java/) Creates an enum set with the same element type as the specified enum set, initially containing all the elements of this type that are not contained in the specified set.
* [EnumSet allOf(Class elementType):](https://www.geeksforgeeks.org/enumset-allof-method-in-java/) Creates an enum set containing all of the elements in the specified element type.
* [EnumSet range(E from, E to):](https://www.geeksforgeeks.org/enumset-range-method-in-java/) Creates an enum set initially containing all of the elements in the range defined by the two specified endpoints.
* [EnumSet copyof():](https://www.geeksforgeeks.org/enumset-copyof-method-in-java/) The method is used to copy all of the contents from a collection to a new enum set.
* [EnumSet copyof(EnumSet s):](https://www.geeksforgeeks.org/enumset-copyof-method-in-java/) The method is used to copy all of the contents from an existing EnumSet to a new enum set.
* [EnumSet clone():](https://www.geeksforgeeks.org/enumset-clone-method-in-java/) The method is used to return a shallow copy of the existing or this set.
* [EnumSet noneOf():](https://www.geeksforgeeks.org/enumset-noneof-method-in-java/) The method is used to create a null set.

When we plan to use an *EnumSet* we have to take into consideration some important points:

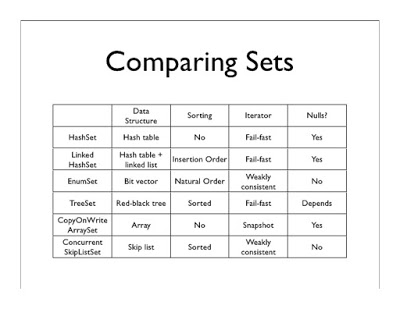
* **It can contain only *enum* values** and all the values have to belong to the same *enum*
* **It doesn’t allow to add null values**, throwing a *NullPointerException* in an attempt to do so
* **It’s not thread-safe**, so we need to synchronize it externally if required
* **The elements are stored following the order in which they are declared in the *enum***
* **It uses a fail-safe iterator** that works on a copy, so it won’t throw a *ConcurrentModificationException* if the collection is modified when iterating over it

**Benefits from Using an EnumSet**

* Due to the implementation of an EnumSet that we’ve described above, all the methods in an EnumSet are implemented using arithmetic bitwise operations. These computations are very fast and therefore all the basic operations are executed in a constant time.
* If we compare EnumSet with other Set implementations like HashSet, the first is usually faster because the values are stored in a predictable order and only one bit needs to be examined for each computation. Unlike HashSet, there’s no need to compute the hashcode to find the right bucket.
* Moreover, due to the nature of bit vectors, an EnumSet is very compact and efficient. Therefore, it uses less memory, with all the benefits that it brings.

### Important properties of EnumSet in Java

Like any other Collection class, EnumSet also has some special properties, which help you to decide when to use EnumSet in your Java application. Keep a note of these properties, behaviours:  
  
1) EnumSet is a special Set implementation, only applicable for [Enums in Java](http://javarevisited.blogspot.sg/2012/12/how-to-create-thread-safe-singleton-in-java-example.html), but you can only store instances of the single enum type. Adding an instance of different enum will result in compile time error, as EnumSet provide type-safety.  
  
2) EnumSet internal representation is extremely efficient and represented as bit vectors. Library itself chooses one of two implementations available depending upon the size of a key universe. RegularEnumSet has chosen if a number of instances are less than 64, otherwise JumboEnumSet is used.  
  
3) As described in Effective Java, EnumSet can be safely used as a type-safe alternative of traditional int based "bit flags". EnumSet provides much-needed readability, without compromising performance.  
  
4) Iterator returned by EnumSet traverse the elements in their natural order, i.e. the order on which enum constants are declared, or the order returned by ordinal() method.  
  
5) An EnumSet [iterator](http://javarevisited.blogspot.sg/2011/10/java-iterator-tutorial-example-list.html) is weakly consistent and never throws ConcurrentModificationException, and may or may not show the effect of any modification to the Set while the iteration is in progress.  
  
6) EnumSet is also not synchronized in Java. Though if you need, you can make EnumSet synchronized similar to other collection by using utility methods from Collections class. Here is how to do that :  
  
Set<YourEnum> s = Collections.synchronizedSet(EnumSet.noneOf(YourEnum.class));  
  
7) EnumSet is an [abstract class](http://javarevisited.blogspot.sg/2013/05/difference-between-abstract-class-vs-interface-java-when-prefer-over-design-oops.html), which means you cannot create its instance using new() operator. This is actually carefully thought to provide special implementation, and that's why EnumSet provides several static factory methods for creating instance e.g. noneOf() returns an empty EnumSet with specified enum type, EnumSet.of(....) returns Set of specified enum constants and allOf()method creates an enum set containing all elements of specified enum.  
  
8) EnumSet also provides methods like complementOf(EnumSet es) and copyOf(EnumSet es) to create EnumSet from other EnumSet, coplementOf() returns enum set with containing all the elements of this type that are not contained in the specified set.



The beauty of EnumSet is that we can perform any collection operation like (add, remove, etc. ) on Enum. As EnumSet is a collection, it provides some Static methods by which we can do union, range check, none, etc. operations.

So by using EnumSet, we can dynamically add Enum elements, remove an element, or perform union operations.

So where can we use EnumSet? Well, one case is role and operation implementation. Not only that, say from the UI, we can manipulate it and create a new role with a new operation set.

To be specific, say I have three types of roles: Self, Friend, and Guest.

As you can guess, the owner of the profile gets the Self role, friends of the owner get the Friend role, and any unknown people get the Guest role. Now our Owner wants to create a new role for a few guests — a role where they can perform Friend operations as well as guest operations.

Implementation

Obviously, we can consider Bit representation for operations and check if a particular operation is valid for a role or not — the same way we can create a new role. I don’t go for that implementation, though. That will be very hard to maintain. Rather I will try to solve it through an EnumSet.

1. I create an Operation Enum where I store all the possible Operations.

package com.example.enumtest;

public enum Operation {

VIEW\_PROFILE,EDIT\_PROFILE,DELETE\_PROFILE,VIEW\_ALBUM,EDIT\_ALBUM,DELETE\_ALBUM,COMMENT,RATE\_PROFILE

}

2. Then I create a Role Enum and associate permitted operations with those roles.

package com.example.enumtest;

import java.util.EnumSet;

import com.sun.xml.internal.ws.policy.privateutil.PolicyUtils.Collections;

public enum Role {

SELF(EnumSet.of(Operation.VIEW\_PROFILE,Operation.VIEW\_ALBUM,Operation.EDIT\_PROFILE,Operation.EDIT\_ALBUM,Operation.DELETE\_PROFILE,

Operation.DELETE\_ALBUM,Operation.COMMENT)),

FRIEND(EnumSet.of(Operation.VIEW\_ALBUM,Operation.VIEW\_PROFILE,Operation.COMMENT)),

GUEST(EnumSet.of(Operation.VIEW\_PROFILE,Operation.RATE\_PROFILE));

EnumSet<Operation> operationSet;

Role(EnumSet<Operation> operationSet){

this.operationSet=operationSet;

}

}

Look in my Role Constructor — I pass EnumSet<Operation>, and when we define the Enum element, I called a Static method on EnumSet. That will perform a union operation.

3. Now, each role has its set of operations, and we can easily print them by traversing operationSet, easily checking whether an operation is permitted:

operationSet.contains(operation)

4. Now I want to add a new Role that can perform both Friend and Guest operations.

package com.example.enumtest;

import java.util.EnumSet;

public class ProfileManager {

public static void printOperationForCustomRole(EnumSet<Role>roles)

{

EnumSet<Operation> mergeSet = EnumSet.noneOf(Operation.class);

for(Role role: roles)

{

for(Operation op : role.operationSet)

mergeSet.add(op);

}

System.out.println(mergeSet);

}

public static void main(String[] args) {

ProfileManager.printOperationForCustomRole(EnumSet.of(Role.FRIEND,Role.GUEST));

}

}

Look how easily we can do it. Just pass Role.Friend and Role.Guest in the printOperationForCustomRole() role method and create a new Operation set that holds all the operations of Friend and Guest.

And for our output:

[VIEW\_PROFILE, VIEW\_ALBUM, COMMENT, RATE\_PROFILE]

**EnumSet and EnumMap**

[EnumSet](https://docs.oracle.com/javase/8/docs/api/java/util/EnumSet.html) and [EnumMap](https://docs.oracle.com/javase/8/docs/api/java/util/EnumMap.html) are compact, efficient implementations of the [Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html) and [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html) interfaces. They have the constraint that their elements/keys come from a single [enum](https://docs.oracle.com/javase/tutorial/java/javaOO/enum.html) type.

Like [HashSet](https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html) and [HashMap](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html), they are modifiable.

In contrast to HashSet, EnumSet:

* Consumes less memory, usually.
* Is faster at all the things a Set can do, usually.
* Iterates over elements in a predictable order (the declaration order of the element type's enum constants).
* Rejects null elements.

In contrast to HashMap, EnumMap:

* Consumes less memory, usually.
* Is faster at all the things a Map can do, usually.
* Iterates over entries in a predictable order (the declaration order of the key type's enum constants).
* Rejects null keys.

If you're wondering how this is possible, I encourage you to look at the source code:

* [EnumSet](http://hg.openjdk.java.net/jdk10/jdk10/jdk/file/72f33dbfcf3b/src/java.base/share/classes/java/util/EnumSet.java)
* A bit vector of the [ordinals](https://docs.oracle.com/javase/8/docs/api/java/lang/Enum.html#ordinal--) of the elements in the Set. This is an abstract superclass of RegularEnumSet and JumboEnumSet.
* [RegularEnumSet](http://hg.openjdk.java.net/jdk10/jdk10/jdk/file/72f33dbfcf3b/src/java.base/share/classes/java/util/RegularEnumSet.java)
* An EnumSet whose bit vector is a single primitive long, which is enough to handle all enum types having 64 or fewer constants.
* [JumboEnumSet](http://hg.openjdk.java.net/jdk10/jdk10/jdk/file/72f33dbfcf3b/src/java.base/share/classes/java/util/JumboEnumSet.java)
* An EnumSet whose bit vector is a long[] array, which is allocated however many slots are necessary for the given enum type. Two slots are allocated for 128 or fewer constants, three slots for 192 or fewer constants, etc.
* [EnumMap](http://hg.openjdk.java.net/jdk10/jdk10/jdk/file/72f33dbfcf3b/src/java.base/share/classes/java/util/EnumMap.java)
* A flat array of the Map's values indexed by the ordinals of their keys.

EnumSet and EnumMap cheat! They use [privileged code like this](http://hg.openjdk.java.net/jdk10/jdk10/jdk/file/72f33dbfcf3b/src/java.base/share/classes/java/util/EnumSet.java#l405):

/\*\* \* 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);}

If you want all the [Month](https://docs.oracle.com/javase/8/docs/api/java/time/Month.html) constants, you might call [Month.values()](https://docs.oracle.com/javase/8/docs/api/java/time/Month.html#values--), giving you a Month[] array. There is a single backing array instance of those Monthconstants living in memory somewhere ([a private field in the Class object](http://hg.openjdk.java.net/jdk10/jdk10/jdk/file/72f33dbfcf3b/src/java.base/share/classes/java/lang/Class.java#l3447) for Month), but it wouldn't be safe to pass that array directly to every caller of values(). Imagine if someone modified that array! Instead, values() creates a fresh clone of the array for each caller.

EnumSet and EnumMap get to skip that cloning step. They have direct access to the backing array.

Effectively, no third-party versions of these classes can be as efficient. Third-party libraries that provide enum-specialized collections tend to delegate to EnumSetand EnumMap. It's not that the library authors are lazy or incapable; delegating is the correct choice for them.

## *When should they be used?*

Historically, Enum{Set,Map} were recommended as a matter of safety, taking better advantage of Java's type system than the alternatives.

**Prefer enum types and Enum{Set,Map} over int flags.**

[Effective Java](https://www.amazon.com/Effective-Java-2nd-Joshua-Bloch/dp/0321356683) goes into detail about this use case for Enum{Set,Map} and enumtypes in general. If you write a lot of Java code, then you should read that book and follow its advice.

Before enum types existed, people would declare flags as int constants. Sometimes the flags would be powers of two and combined into sets using bitwise arithmetic:

static final int OVERLAY\_STREETS = 1 << 0;static final int OVERLAY\_ELECTRIC = 1 << 1;static final int OVERLAY\_PLUMBING = 1 << 2;static final int OVERLAY\_TERRAIN = 1 << 3;void drawCityMap(int overlays) { ... }drawCityMap(OVERLAY\_STREETS | OVERLAY\_PLUMBING);

Other times the flags would start at zero and count up by one, and they would be used as array indexes:

static final int MONSTER\_SLIME = 0;static final int MONSTER\_GHOST = 1;static final int MONSTER\_SKELETON = 2;static final int MONSTER\_GOLEM = 3;int[] kills = getMonstersSlain();if (kills[MONSTER\_SLIME] >= 10) { ... }

These approaches got the job done for many people, but they were somewhat error-prone and difficult to maintain.

When enum types were introduced to the language, Enum{Set,Map} came with them. Together they were meant to provide better tooling for problems previously solved with int flags. We would say, "Don't use int flags, use enum constants. Don't use bitwise arithmetic for sets of flags, use EnumSet. Don't use arrays for mappings of flags, use EnumMap." This was not because the enum-based solutions were faster than int flags — they were probably slower — but because the enum-based solutions were easier to understand and implement correctly.

Fast forward to today, I don't see many people using int flags anymore (though there are [notable exceptions](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#characteristics--)). We've had enum types in the language for more than a decade. We're all using enum types here and there, we're all using the collections framework. At this point, while *Effective Java*'s advice regarding Enum{Set,Map} is still valid, I think most people will never have a chance to put it into practice.

Today, we're using enum types in the right places, but we're forgetting about the collection types that came with them.

**Prefer Enum{Set,Map} over Hash{Set,Map} as a performance optimization.**

* Prefer EnumSet over HashSet when the elements come from a single enumtype.
* Prefer EnumMap over HashMap when the keys come from a single enum type.

Should you refactor all of your existing code to use Enum{Set,Map} instead of Hash{Set,Map}? **No.**

Your code that uses Hash{Set,Map} isn't wrong. Migrating to Enum{Set,Map}might make it faster. That's it.

If you've ever used primitive collection libraries like [fastutil](http://fastutil.di.unimi.it/) or [Trove](http://trove.starlight-systems.com/), then it may help to think of Enum{Set,Map} like those primitive collections. The difference is that Enum{Set,Map} are specialized for enum types, not primitive types, and you can use them without depending on any third-party libraries.

Enum{Set,Map} don't have identical semantics to Hash{Set,Map}, so please don't make blind, blanket replacements in your existing code.

Instead, try to remember these classes for next time. If you can make your code more efficient for free, then why not go ahead and do that, right?

If you use [IntelliJ IDEA](https://www.jetbrains.com/idea/), you can have it remind you to use Enum{Set,Map} with inspections:

* Analyze - Run inspection by name - "Set replaceable with EnumSet" or "Map replaceable with EnumMap"

...or...

* File - Settings - Editor - Inspections - Java - Performance issues - "Set replaceable with EnumSet" or "Map replaceable with EnumMap"

[SonarQube](https://www.sonarqube.org/) can also remind you to use Enum{Set,Map}:

* [S1641](https://github.com/SonarSource/sonar-java/blob/cb1baeadb6f9beaab8a04a639a33b90944f73f19/java-checks/src/main/java/org/sonar/java/checks/EnumSetCheck.java): "Sets with elements that are enum values should be replaced with EnumSet"
* [S1640](https://github.com/SonarSource/sonar-java/blob/cb1baeadb6f9beaab8a04a639a33b90944f73f19/java-checks/src/main/java/org/sonar/java/checks/EnumMapCheck.java): "Maps with keys that are enum values should be replaced with EnumMap"

For immutable versions of Enum{Set,Map}, see the following methods from [Guava](https://github.com/google/guava):

* Factory methods:
  + [Sets.immutableEnumSet(first, rest...)](http://google.github.io/guava/releases/21.0/api/docs/com/google/common/collect/Sets.html#immutableEnumSet-E-E...-)
  + [Sets.immutableEnumSet(iterable)](http://google.github.io/guava/releases/21.0/api/docs/com/google/common/collect/Sets.html#immutableEnumSet-java.lang.Iterable-)
  + [Maps.immutableEnumMap(map)](http://google.github.io/guava/releases/21.0/api/docs/com/google/common/collect/Maps.html#immutableEnumMap-java.util.Map-)
* Collectors:
  + [Sets.toImmutableEnumSet()](http://google.github.io/guava/releases/21.0/api/docs/com/google/common/collect/Sets.html#toImmutableEnumSet--)
  + [Maps.toImmutableEnumMap(keyMapper, valueMapper)](http://google.github.io/guava/releases/21.0/api/docs/com/google/common/collect/Maps.html#toImmutableEnumMap-java.util.function.Function-java.util.function.Function-)
  + [Maps.toImmutableEnumMap(keyMapper, valueMapper, mergeFunction)](http://google.github.io/guava/releases/21.0/api/docs/com/google/common/collect/Maps.html#toImmutableEnumMap-java.util.function.Function-java.util.function.Function-java.util.function.BinaryOperator-)

If you don't want to use Guava, then wrap the modifiable Enum{Set,Map}instances in [Collections.unmodifiableSet(set)](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#unmodifiableSet-java.util.Set-) or [Collections.unmodifiableMap(map)](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html#unmodifiableMap-java.util.Map-) and throw away the direct references to the modifiable collections.

The resulting collections may be less efficient when it comes to operations like containsAll and equals than their counterparts in Guava, which may in turn be less efficient than the raw modifiable collections themselves.

## *Could the implementations be improved?*

Since they can't be replaced by third-party libraries, Enum{Set,Map} had better be as good as possible! They're good already, but they could be better.

Enum{Set,Map} have missed out on potential upgrades since Java 8. New methods were added in Java 8 to Set and Map (or higher-level interfaces like [Collection](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html) and [Iterable](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html)). While the default implementations of those methods are *correct*, we could do better with overrides in Enum{Set,Map}.

This issue is tracked as [JDK-8170826](https://bugs.openjdk.java.net/browse/JDK-8170826).

Specifically, these methods should be overridden:

* {Regular,Jumbo}EnumSet.forEach(action)
* {Regular,Jumbo}EnumSet.iterator().forEachRemaining(action)
* {Regular,Jumbo}EnumSet.spliterator()
* EnumMap.forEach(action)
* EnumMap.{keySet,values,entrySet}().forEach(action)
* EnumMap.{keySet,values,entrySet}().iterator().forEachRemaining(action)
* EnumMap.{keySet,values,entrySet}().spliterator()

I put [sample implementations](https://github.com/TechEmpower/misc/blob/master/enums/enum_collection_overrides.txt) on GitHub in case you're curious what these overrides might look like. They're all pretty straightforward.

Rather than walk through each implementation in detail, I'll share some high-level observations about them.

* The optimized forEach and forEachRemaining methods are roughly 50% better than the defaults (in terms of operations per second).
* EnumMap.forEach(action) benefits the most, becoming twice as fast as the default implementation.
* The [iterable.forEach(action)](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html) method is popular. Optimizing it tends to affect a large audience, which increases the likelihood that the optimization (even if small) is worthwhile. (I'd claim that iterable.forEach(action) is *too* popular, and I'd suggest that the traditional enhanced for loop should be preferred over forEach except when the argument to forEach can be written as a method reference. That's a topic for another discussion, though.)
* The [iterator.forEachRemaining(action)](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html#forEachRemaining-java.util.function.Consumer-) method is more important than it seems. Few people use it directly, but many people use it indirectly through streams. The default spliterator() delegates to the iterator(), and the default stream() delegates to the spliterator(). In the end, stream traversal may delegate to iterator().forEachRemaining(...). Given the popularity of streams, optimizing this method is a good idea!
* The [iterable.spliterator()](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html#spliterator--) method is critical when it comes to stream performance, but writing a custom [Spliterator](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html) from scratch is a non-trivial task. I recommend this approach:
  + Check whether the characteristics of the default spliterator are correct for your collection (often times the defaults are too conservative — for example, EnumSet's spliterator is currently missing the [ORDERED](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#ORDERED), [SORTED](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#SORTED), and [NONNULL](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#NONNULL) characteristics). If they're not correct, then provide a trivial override of the spliterator that uses [Spliterators.spliterator(collection, characteristics)](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterators.html#spliterator-java.util.Collection-int-)to define the correct characteristics.
  + Don't go further than that until you've read through [the implementation of that spliterator](http://hg.openjdk.java.net/jdk10/jdk10/jdk/file/72f33dbfcf3b/src/java.base/share/classes/java/util/Spliterators.java#l1691), and you understand how it works, and you're confident that you can do better. In particular, your [tryAdvance(action)](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#tryAdvance-java.util.function.Consumer-) and [trySplit()](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html#trySplit--) should both be better. Write a benchmark afterwards to confirm your assumptions.
* The [map.forEach(action)](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html#forEach-java.util.function.BiConsumer-) method is extremely popular and is almost always worth overriding. This is especially true for maps like EnumMap that create their [Entry](https://docs.oracle.com/javase/8/docs/api/java/util/Map.Entry.html) objects on demand.
* It's usually possible to share code across the forEach and forEachRemaining methods. If you override one, you're already most of the way there to overriding the others.
* I don't think it's worthwhile to override [collection.removeIf(filter)](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html#removeIf-java.util.function.Predicate-)in any of these classes. For RegularEnumSet, where it seemed most likely to be worthwhile, I couldn't come up with a faster implementation than the default.
* Enum{Set,Map} *could* provide faster hashCode() implementations than the ones they currently inherit from [AbstractSet](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractSet.html) and [AbstractMap](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractMap.html), but I don't think that would be worthwhile. In general, I don't think optimizing the hashCode() of collections is worthwhile unless it can somehow become a constant-time (O(1)) operation, and even then it is questionable. Collection hash codes aren't used very often.