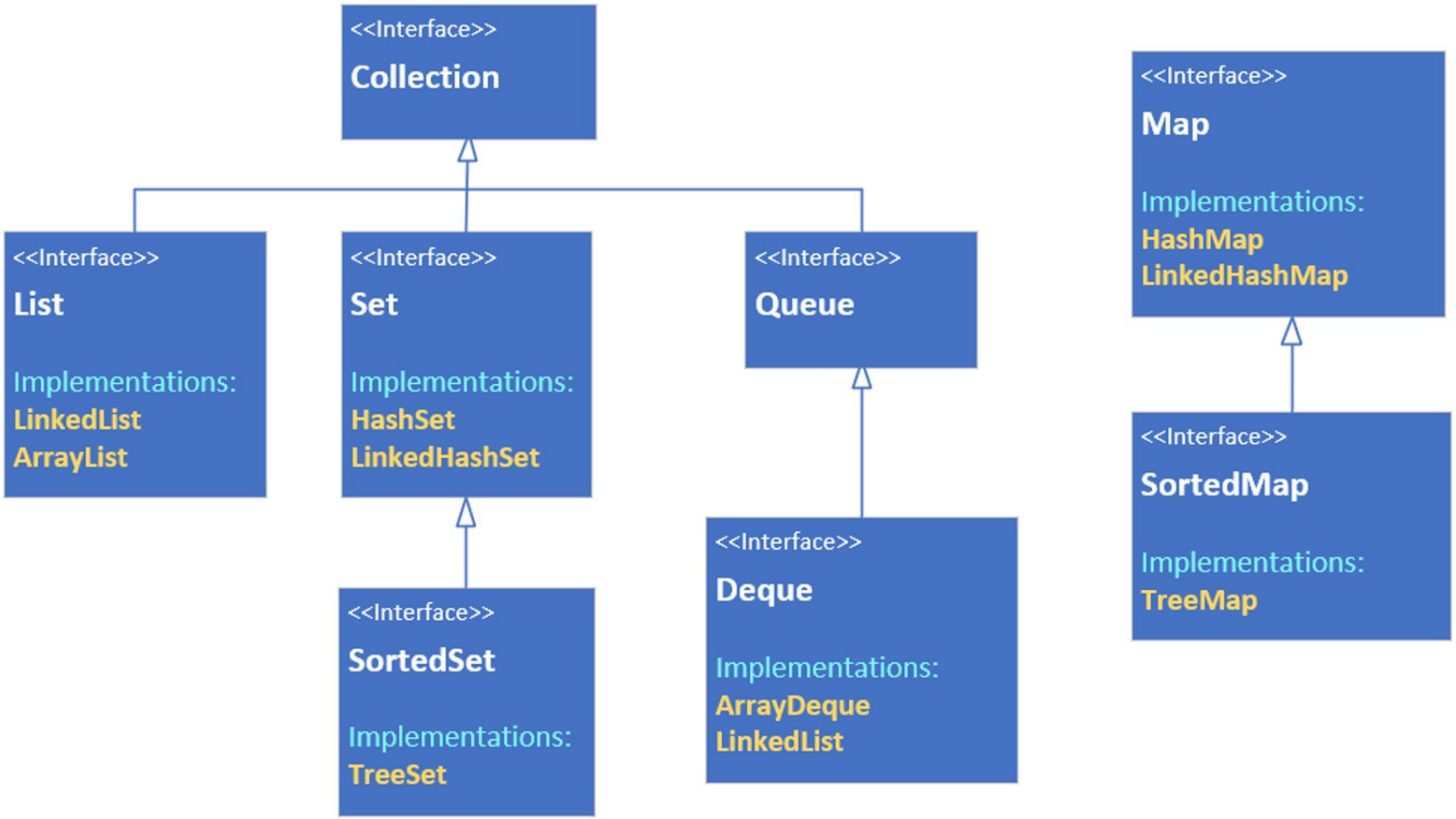
NPTEL

noc24_cs43

Programming in Java

Live Interaction Session 12: 09th Apr 2024

The Map Interface, why is it different?



The Map Interface, why is it different?

A map in the collections framework is another data structure.

Although it's still a grouping of elements, it's different, because elements are stored with keyed references.

This means a Map requires two type arguments, as you can see on this slide, where I'm showing the root interface, Collection, compared to the Map interface.

Collection Interface	Map Interface
`interface Collection<E> extends Iterable<E>`	`interface Map<K, V>`

The Map has K for it's key type, and V for the value type.

As with any generic classes, the only restriction on these types is, they must be reference types, and not primitives.

Map characteristics

A Java Map can't contain duplicate keys.

Each key can only map to a single value.

The Java classes that implement the map interface, the **HashMap**, the **LinkedHashMap**, and the **TreeMap**.

The HashMap is unordered, the LinkedHashMap is ordered by insertion order, and the TreeMap is a sorted map.

Ordered and Sorted Map implementations

The Map interface has the **LinkedHashMap** and **TreeMap** classes.

The LinkedHashMap is a key value entry collection, whose keys are ordered by insertion order.

The TreeMap is sorted by it's keys, so a key needs to implement Comparable, or be initialized, with a specified Comparator.

What's a view?

The view, or view collection as Java calls it, doesn't store elements, but depends on a backing collection that stores the data elements.

You saw this with the headSet, tailSet and subSet methods on Sets.

You're also very familiar now, with a list backed by an array, a view we get back, when we use the Arrays.asList method, to get an array in the form of a list.

You'll remember when we make changes to that list, the changes are reflected in the underlying array, and vice versa.

The functionality available to us on the list, is limited to features supported by the backing storage, so for a list backed by an array, we can't add or remove elements as an example.

The purpose of view collections

Some of you might be familiar with database views which hide the details of the underlying data structures, to make it easier for clients to use the data.

These view collections serve a similar purpose.

They let us manipulate the collections, without really having to know exact details, about the storage of the data.

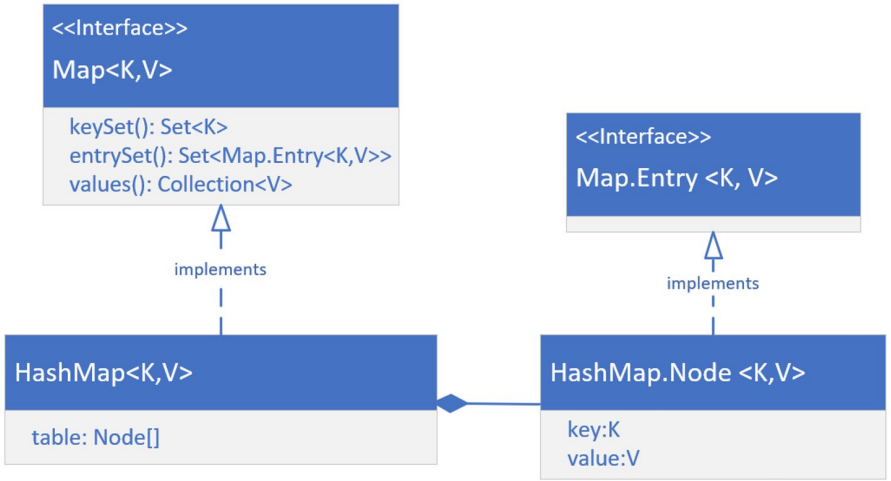
In other words, we don't have to keep learning new methods, to manipulate data.

As long as we can get a collection view of the data, we can use many of the collection methods, to simplify our work.

The HashMap's implementation

On this slide, I'm showing you a high level overview of the structure of the HashMap class.

First, it's important to know that there's a static nested interface on the Map interface, and it's name is Entry.



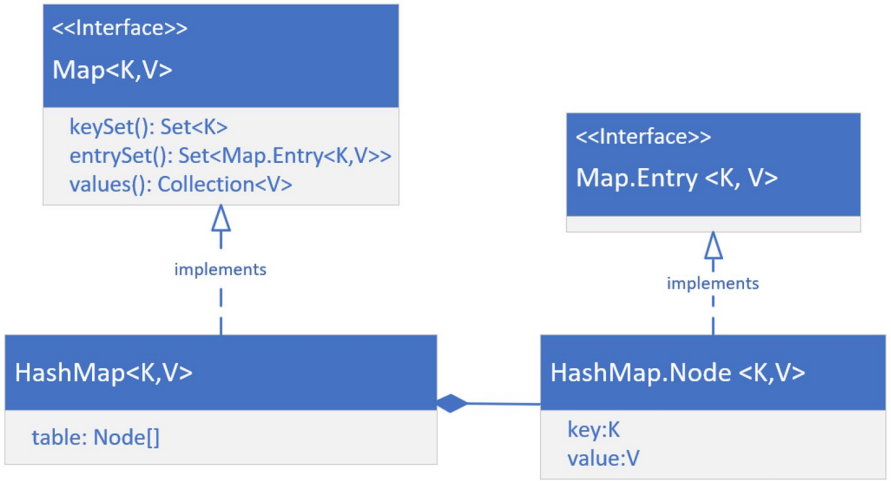
The HashMap's implementation

Concrete classes, implement both the Map interface, and the Map.Entry interface.

The HashMap implements Map, and has a static nested class, Node, that implements the Map.Entry interface.

The HashMap maintains an array of these Nodes, in a field called table, whose size is managed by Java, and whose indices are determined by hashing functions.

For this reason, a HashMap is not ordered.



The Map's view collections

There are three view collections we can get from the map, which are the key set, the entry set, and the values.

We know a map has keys, and these can't contain duplicates.

These keys can be retrieved as a Set view, by calling the keySet method on any map object.

Each key value pair is stored as an instance of an Entry, and the combination of the key and value will be unique, because the key is unique.

The Map's view collections

A Set view of these entries, or nodes in the case of the HashMap, can be retrieved from the method entrySet.

Finally, values are stored, and referenced by the key, and values can have duplicates, meaning multiple keys could reference a single value.

You can get a Collection view of these, from the values method, on a map instance.

Functionality available to set returned from keySet()

The set returned from the **keySet** method, is backed by the map.

This means changes to the map are reflected in the set, and vice-versa.

The set supports element removal, which removes the corresponding mapping from the map.

You can use the methods **remove**, **removeAll, retainAll**, and **clear**.

It does not support the **add** or **addAll** operations.

EnumSet and EnumMap

There are two more classes in the collections framework, specifically created to support enum types more efficiently.

You can use any List, Set, or Map, with an enum constant.

The EnumSet, and EnumMap, each has a special implementation that differs from the HashSet or HashMap.

These implementations make these two types extremely compact and efficient.

There's no special list implementation for enum types.

The EnumSet

The EnumSet is a specialized Set implementation for use with enum values.

All of the elements in an EnumSet must come from a single enum type.

The EnumSet is abstract, meaning we can't instantiate it directly.

It comes with many factory methods to create instances.

In general, this set has much better performance than using a HashSet, with an enum type.

Bulk operations (such as containsAll and retainAll) should run very quickly, in constant time, O(1), if they're run on an enumSet, and their argument is an EnumSet.

The EnumMap

The Enum Map is a specialized Map implementation for use with enum type keys.

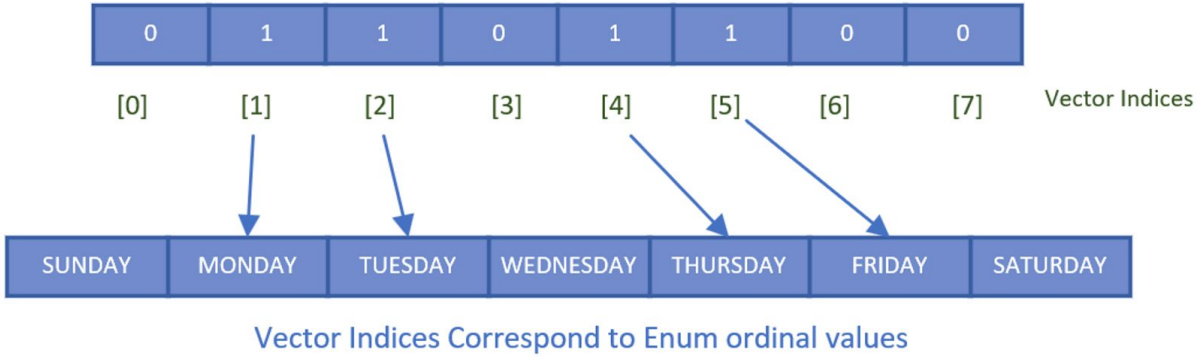
The keys must all come from the same enum type, and they're ordered naturally by the ordinal value of the enum constants.

This map has the same functionality as a HashMap, with O(1) for basic operations.

The enum key type is specified during construction of the EnumMap, either explicitly by passing the key type's class, or implicitly by passing another EnumSet.

In general, this map has better performance than using a HashMap, with an enum type.

Eashaan Work Day EnumSet

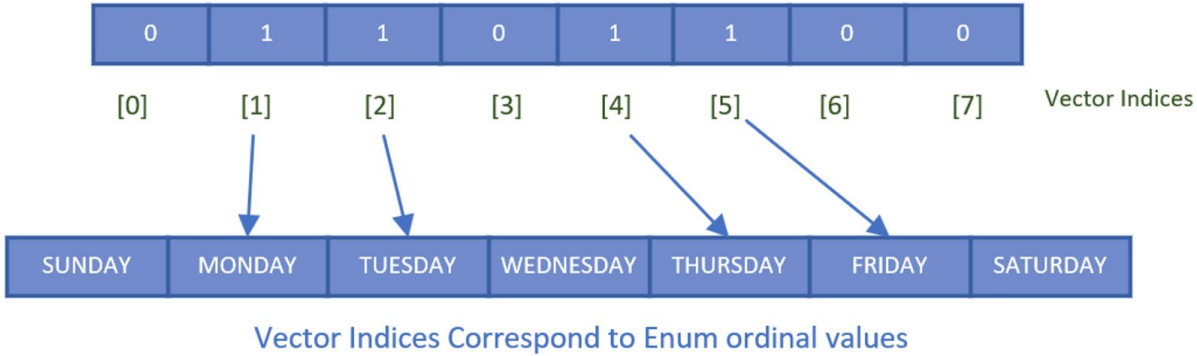


This slide is a visual representation of the EnumSet for Eashaan work days.

It's size is 7, for the 7 possible values (based on the number of constants in the WeekDay Enum).

Any weekday that's part of her set, will be set to 1, at the index that corresponds to, the weekday ordinal value.

Eashaan Work Day EnumSet



MONDAY has an ordinal value of 1 in our WeekDays enum, and the value in the underlying bit vector, at position 1, is a 1.

This means MONDAY is part of Eashaan EnumSet.

All Collections Operations:

For Lists (e.g., ArrayList, LinkedList):

- **add(E e)**: Adds an element to the end of the list.
- **addAll(Collection<? extends E> c)**: Adds all elements from a collection to the end of the list.
- **get(int index)**: Retrieves the element at the specified index.
- **remove(int index)**: Removes the element at the specified index.
- **size()**: Returns the number of elements in the list.
- **clear()**: Removes all elements from the list.
- **contains(Object o)**: Returns true if the list contains the specified element.
- **isEmpty()**: Returns true if the list contains no elements.
- **indexOf(Object o)**: Returns the index of the first occurrence of the specified element in the list, or -1 if the list does not contain the element.
- **subList(int fromIndex, int toIndex)**: Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

All Collections Operations:

For Sets (e.g., HashSet, TreeSet):

- **add(E e)**: Adds the specified element to the set if it is not already present.
- **addAll(Collection<? extends E> c)**: Adds all of the elements in the specified collection to the set if they're not already present.
- **remove(Object o)**: Removes the specified element from the set if it is present.
- **contains(Object o)**: Returns true if the set contains the specified element.
- **isEmpty()**: Returns true if the set contains no elements.
- **size()**: Returns the number of elements in the set.
- **clear()**: Removes all elements from the set.

All Collections Operations:

For Maps (e.g., HashMap, TreeMap):

- **put(K key, V value)**: Associates the specified value with the specified key in the map.
- **get(Object key)**: Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
- **remove(Object key)**: Removes the mapping for a key from this map if it is present.
- **containsKey(Object key)**: Returns true if this map contains a mapping for the specified key.
- **containsValue(Object value)**: Returns true if this map maps one or more keys to the specified value.
- **keySet()**: Returns a Set view of the keys contained in this map.
- **values()**: Returns a Collection view of the values contained in this map.
- **entrySet()**: Returns a Set view of the mappings contained in this map.

All Collections Operations:

For Enums:

- **values()**: Returns an array containing the constants of this enum type, in the order they're declared.
- **valueOf(String name)**: Returns the enum constant with the specified name.
- **ordinal()**: Returns the ordinal of this enumeration constant (its position in its enum declaration, where the initial constant is assigned an ordinal of zero).