Java Training
Collections Part 2

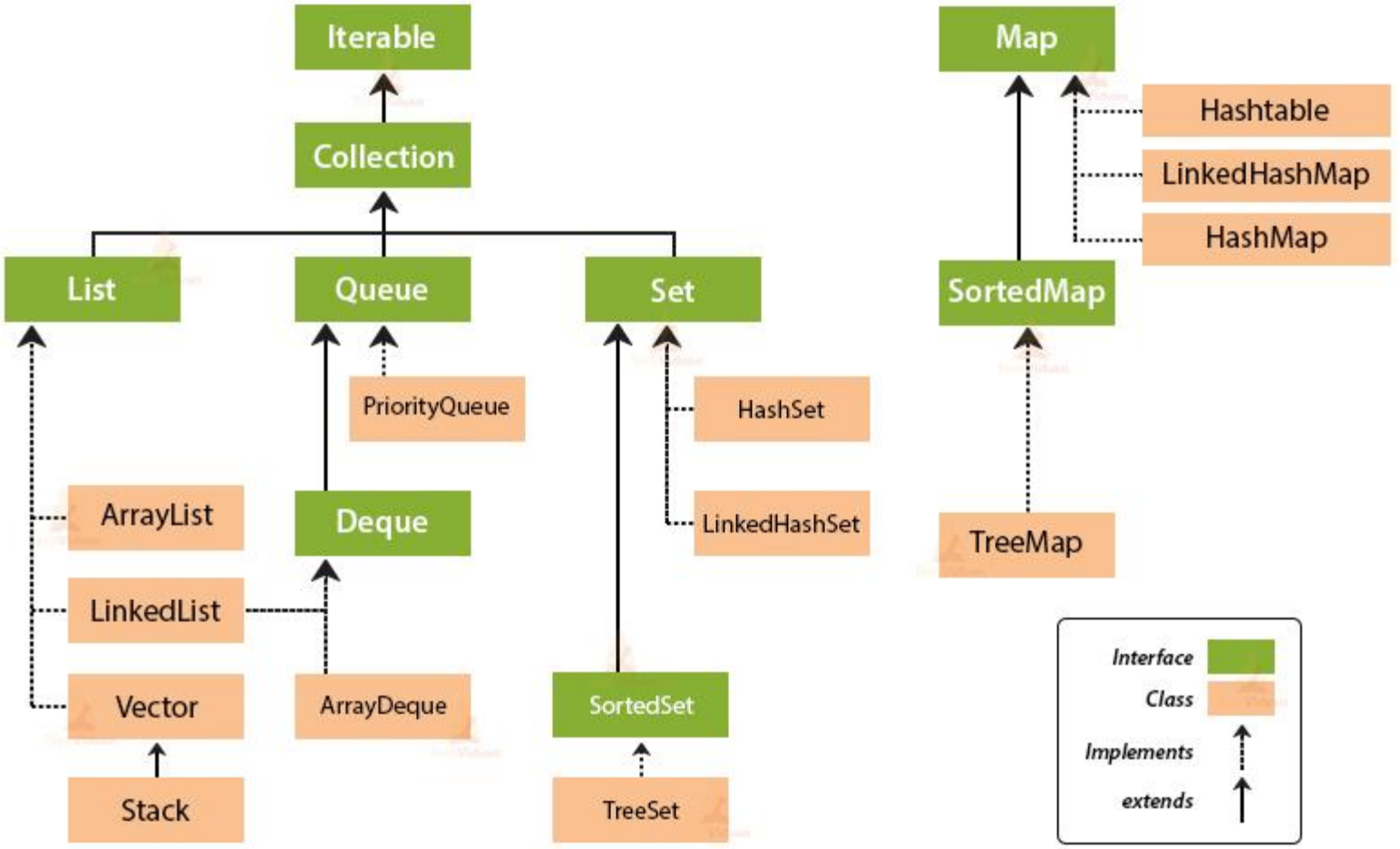
About me

- Lead Software Engineer
- Working for EPAM more than 12 years
- Current position - Development Team Lead

Today’s agenda

- Map interface
- Collection framework – algorithms

Revision

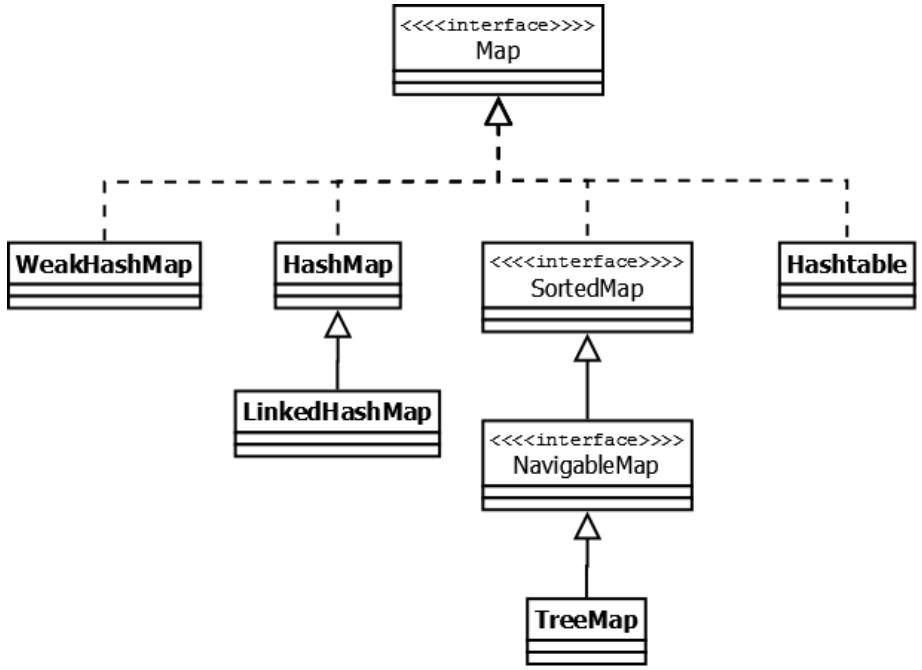


java.util package

Map interface

java.util.Map - an interface that represents a mapping between a **key** and a **value**.

- Is **not** a subtype of the Collection interface
- Contains unique keys
- Doesn't allow duplicate keys, but you can have duplicate values



Method	Description
V put(Object key, Object value)	Insert an entry in the map
V get(Object key)	Returns the object that contains the value associated with the key
boolean containsKey(Object key)	Returns true if some key equal to the key exists within the map, else return false
boolean containsValue(Object value)	returns true if some value equal to the value exists within the map, else return false
int size()	Returns the number of entries in the map

+ other methods

HashMap

HashMap implements Map interface which allows us to store **key** and **value** pair

- Contains values based on the key
- Contains only unique keys
- Allows to have one null key and multiple null values
- Maintains no order
- **Non-synchronized** (**Hashtable** is synchronized)

Create object:

```
Map<Integer, String> hMap = new HashMap <Integer, String>();
```

Add element:

```
hMap.add(4, "Value1");
```

Get element:

```
hMap.get(4);
```

HashMap -internal structure

- https://medium.com/art-of-coding/hash-map-and-its-implementation-in-java-b2bb82fb0309
- https://www.geeksforgeeks.org/internal-working-of-hashmap-java/

LinkedHashMap

LinkedHashMap is Hashtable and Linked list implementation of the Map interface

- Contains values based on the key
- Contains only unique keys
- Allows to have one null key and multiple null values
- Maintains insertion order
- **Non-synchronized**

Create object:

```
Map<Integer, String> lhMap = new LinkedHashMap <Integer,
String>();
```

Add element:

```
lhMap.add(4, "Value1");
```

Get element:

```
lhMap.get(4);
```

TreeMap

TreeMap is a red-black tree based implementation

- Contains values based on the key
- Contains only unique keys
- **Cannot** have a null key but can have multiple null values
- Maintains ascending order (be default)
- **Non-synchronized**

Create object:

```
Map<Integer, String> tMap = new TreeMap <Integer, String>();
```

Add element:

```
tMap.add(4, "Value2");
tMap.add(6, "Value1");
```

Get element:

```
tMap.get(4);
```

Benefits

- **Reduces programming effort.** Algorithms and data structures are already implemented, developer can focus on other important parts of application.
- **Increases program speed and quality.** Programmer does not need to think of the best implementation of a specific data structure. Programmer can simply use the best implementation to drastically boost the performance of algorithm/program.
- **Consistent API.**
- **Reduces effort to design new APIs.** You don't have to reinvent the wheel each time are creating an API that relies on collections; instead, just use standard collection interfaces.
- **Fosters software reuse.** Collections interfaces and implementations are flexible enough to be reused given that there is a contract of interface.

Algorithms

Java collection framework defines several algorithms as static methods that can be used with collections and map objects. These methods are polymorphic implementation of some common algorithms frequently required by the programmer.

- **Sorting**
- **Shuffling**
- **Routine Data Manipulation**
- **Searching**
- **Composition**
- **Finding Extreme Values**

Algorithms: Sorting

The ***sort*** method in the Collections class sorts (according to its elements' *natural ordering*) a collection that implements the **List** interface

Method	Description
void sort(List list)	Sorts the elements of the list as determined by their natural ordering
void sort(List list, Comparator comp)	Sorts the elements of the list as determined by Comparator comp

The sort operation uses a slightly optimized merge sort algorithm that is **fast** and **stable:**

- **Fast -** it is guaranteed to run in n log(n) time and runs substantially faster on nearly sorted lists
- **Stable - i**t doesn't reorder equal elements. This is important if you sort the same list repeatedly on different attributes

Algorithms: Shuffling

Provides a random permutation of elements in the **List**

Method	Description
void shuffle(List list)	Shuffles the elements in list
void shuffle(List list, Random r)	Shuffles the elements in the list by using r as a source of random numbers

Algorithms: Routine Data Manipulation

The Collections class provides five algorithms for doing routine data manipulation on List objects

Method	Description
void reverse(List list)	Reverses all the elements sequence in list
void fill(List list, Object obj)	Assigns obj to each element of the list
void copy(List list1, List list2)	Copies the elements of list2 to list1
void swap(List list, int idx1, int idx2)	Exchanges the elements in the list at the indices specified by idx1 and idx2
boolean addAll(Collection<? extends E> coll)	Appends all elements in an array to a collection

Algorithms: Binary Search

The **binarySearch** algorithm searches for a specified element in a sorted **List**

- the list must be sorted in ascending order according to the natural ordering
- very fast

Method	Description
int binarySearch(List list, Object value)	Returns the position of value in the list (must be in the sorted order), or -1 if value is not found
int binarySearch(List list, Object value, Comparator c)	Returns the position of value in the list ordered according to c, or -1 if value is not found

Algorithms: Composition

The frequency and disjoint algorithms test some aspect of the composition of one or more Collections

Method	Description
int frequency(Collection<?> c, Object o)	Returns the count of elements in c that equal the object o
disjoint(Collection<?> c1, Collection<?> c2)	Returns true if the collections have no elements in common

Algorithms: Finding extreme values

The **min** and the **max** algorithms return, respectively, the minimum and maximum element contained in a specified Collection

Method	Description
Object max(Collection c)	Returns the largest element from the collection c as determined by natural ordering
Object max(Collection c, Comparator comp)	Returns the largest element from the collection c as determined by Comparator comp
Object min(Collection c)	Returns the smallest element from the collection c as determined by natural ordering
Object min(Collection c, Comparator comp)	Returns the smallest element from the collection c as determined by Comparator comp

Links

- https://docs.oracle.com/javase/tutorial/collections/interfaces/index.html
- https://docs.oracle.com/javase/tutorial/collections/algorithms/index.html

Thanks