Java Training
Collections Part 1

About me

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

Today’s agenda

- Before Collections
- Interfaces and concrete collections.

Before Collections Framework

- Arrays
- Vectors
- Hashtables

All these collections had no common interface.

Java Collections Framework

The Java Collections Framework (JCF, since JDK 1.2) is a set of classes and interfaces that implement commonly reusable collection data structures.

- Interfaces
- Implementations
- Algorithms

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.

Revision

- Class

A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.

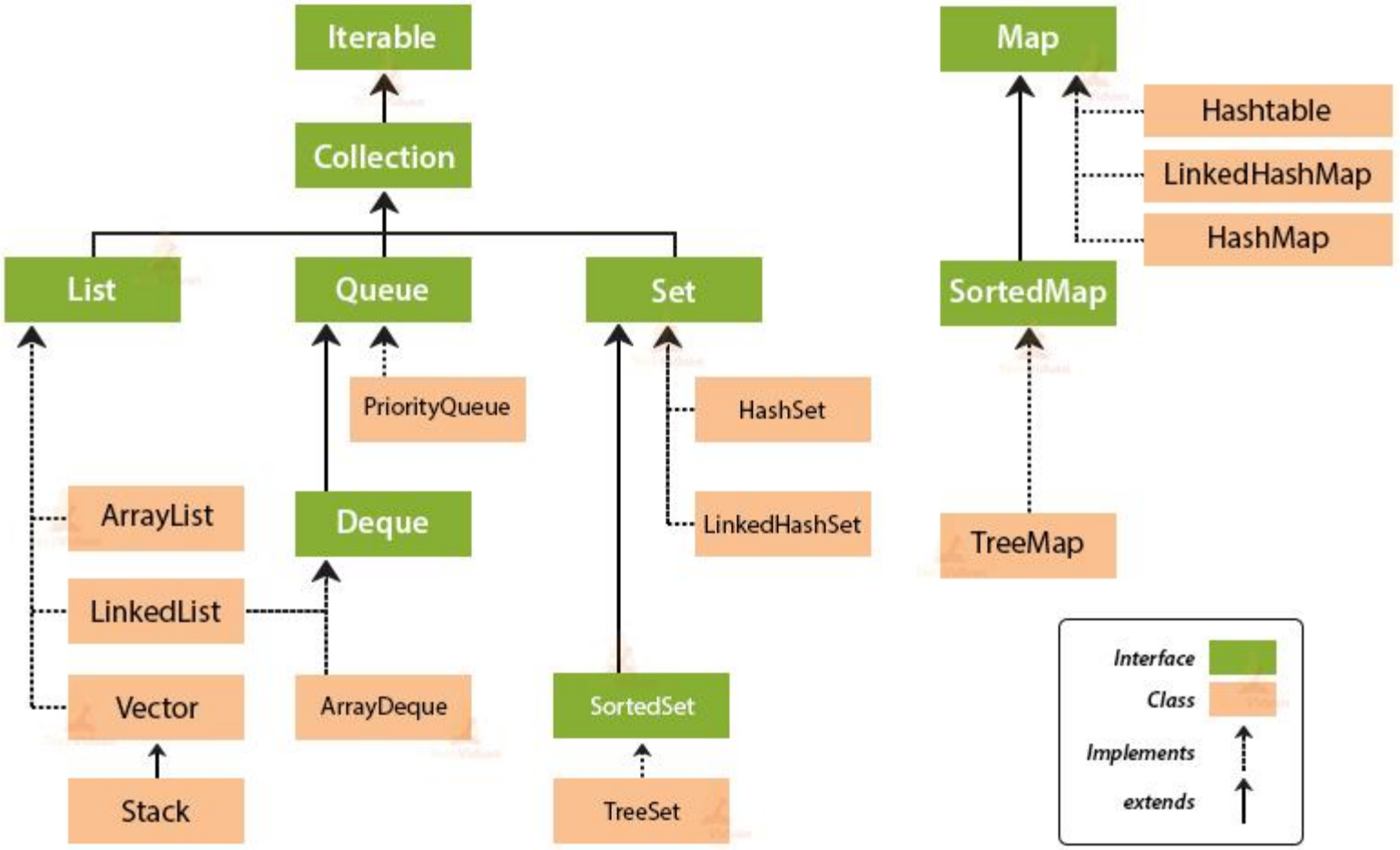
- Interface

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (*only method signature, no body*). Interfaces specify what a class must do and not how. It is the blueprint of the class.

```
interface MyInterface{
    //don't need to implement
    default void newMethod(){
        System.out.println("Newly added default method");
    }
    //don't need to implement + we cannot override it
    static void anotherNewMethod(){
        System.out.println("Newly added static method");
    }
    //we must to implement it
    void existingMethod(String str);
}
```

since java 8

Hierarchy



java.util package

Iterable interface

Iterable interface is the root interface for all the collection classes

Method	Description
Iterator<T> iterator();	Returns the iterator

Iterator interface provides the facility of iterating the elements in a forward direction only.

Method	Description
boolean hasNext()	Returns true if the iterator has more elements otherwise it returns false.
Object next()	Returns the element and moves the cursor pointer to the next element. (throws *NoSuchElementException*)
void remove()	Removes the last elements returned by the iterator.

Collection interface

The **Collection** interface represents a group of objects (elements)

The **Collection** interface is implemented by all the classes in the collection framework

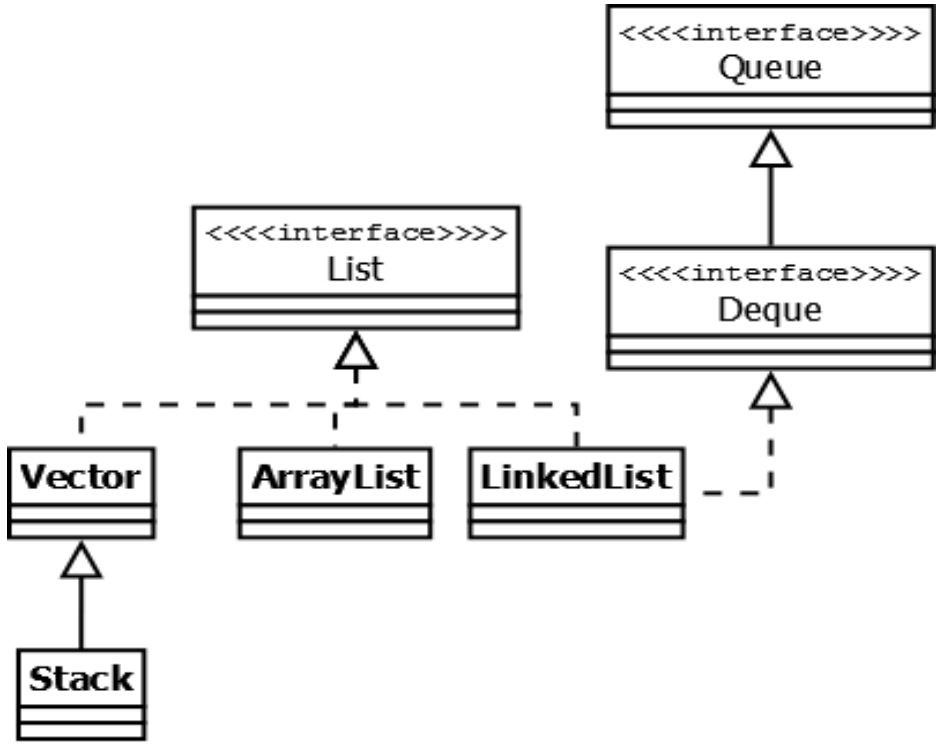
Method	Description
int size() boolean isEmpty();	Returns the number of elements in this collection. true – contains no elements, false – contains.
boolean contains(Object o); boolean containsAll(Collection<?> c)	true – contains the specified element, false – no true – contains all of the elements
boolean add(E e); boolean addAll(Collection<? extends E> c)	Inserts an element to collection Insert the specified collection elements
boolean remove(Object element) boolean removeAll(Collection<?> c) boolean retainAll(Collection<?> c)	Removes the given objection Removes all given objects Retains only given objects

+ other methods

LIST interface

java.util.List - An ordered collection (also known as a sequence)

- Elements have a specific order, and duplicate elements are allowed
- Elements can be placed in a specific position/accessed by integer index
- Elements can be searched for within the list



Method	Description
boolean add(E e);	Appends the specified element to the end of this list.
E get(int index);	Returns the element at the specified position in this list
E set(int index, E element);	Replaces the element at the specified position in this list with the specified element
int indexOf(Object o);	Returns the index of the first occurrence of the specified element
int lastIndexOf(Object o);	Returns the index of the last occurrence of the specified element in this list

+ other methods

ArrayList

ArrayList is resizable-array & default implementation for **List** interface.

- Implements all optional **List** methods, and permits all elements, including null
- Maintains insertion order
- Allows random access because array works at the index basis
- The size of an **ArrayList** is increased automatically if the collection grows or shrinks if the objects are removed from the collection.
- **Non-synchronized** (Vector is synchronized)

Create object:

```
List<String> list = new ArrayList<String>();
```

Add element:

```
list.add("0");
list.add(5, "100");
```

Remove element:

```
list.remove("0");
list.remove(5);
```

LinkedList

Uses a doubly-linked list implementation of the **List** and **Deque.**

- Implements all optional **List/Deque** methods, and permits all elements, including null
- Maintains insertion order
- Manipulation is fast because no shifting needs to occur
- **Non-synchronized**

Create object:

```
List<String> list = new LinkedList<String>();
```

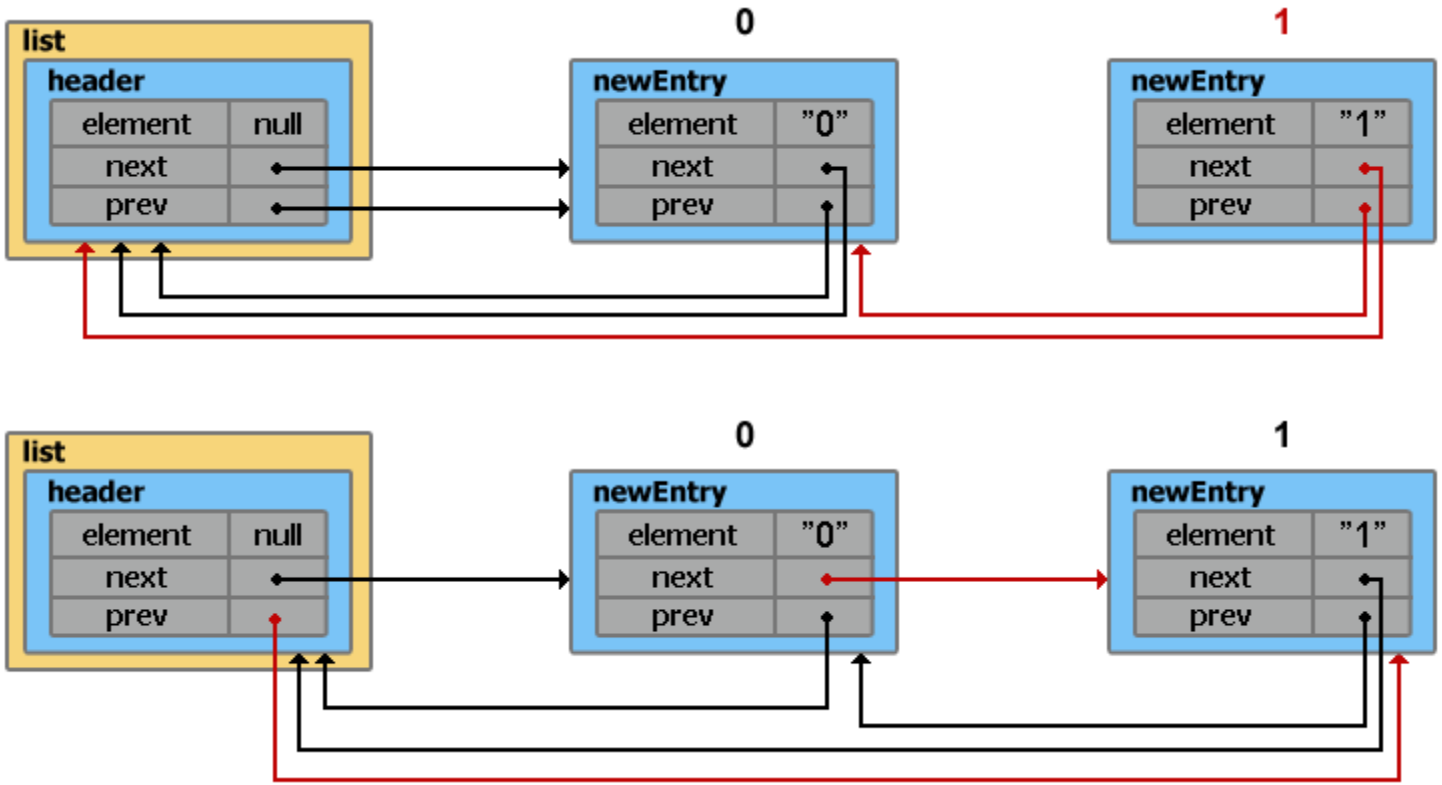
Add element:

```
list.add("0");
list.add(5, "100");
```

Remove element:

```
list.remove("0");
list.remove(5);
```

LinkedList


list
header
element
null
next
prev
0
newEntry
element
"0"
next
prev
1
newEntry
element
"1"
next
prev
list
header
element
null
next
prev
0
newEntry
element
"0"
next
prev
1
newEntry
element
"1"
next
prev

ArrayList vs LinkedList

ArrayList	LinkedList
Stores its elements internally as an array	Stores its elements as a doubly-linked list
Whenever we remove an element, internally, the array is traversed, and the memory bits are shifted.	There is no concept of shifting the memory bits. The list is traversed and the reference link is changed.
Less memory	More memory as it stores next/previous references
List	List + Deque (can be used in Stack/Queue)

Operation	ArrayList	LinkedList
get(int index)	O(1)	O(n)
add(E el) add(int index, E el)	O(1) -> O(n) O(n/2)	O(1) O(n/4)
remove(int index) Iterator.remove()	O(n/2) O(n/2)	O(n/4) O(1)

- **ArrayList** is better for storing and accessing data.
- **LinkedList** is better for manipulating data

Queue interface

java.util.Queue is used to hold the elements to be processed in FIFO order

- Provides additional insertion, removal, and inspection operations.

Method	Description
boolean add(object)	Inserts the specified element into this queue and return true upon success
boolean offer(object)	Insert the specified element into this queue
Object remove()	Retrieves and removes the head of this queue
Object poll()	Retrieves and removes the head of this queue, or returns null if this queue is empty
Object element()	Retrieves, but does not remove, the head of this queue
Object peek()	Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty

PriorityQueue

PriorityQueue elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time.

Create object:

```
Queue<Integer> prQueue = new PriorityQueue< Integer >();
```

Add element:

```
prQueue.add(10);
```

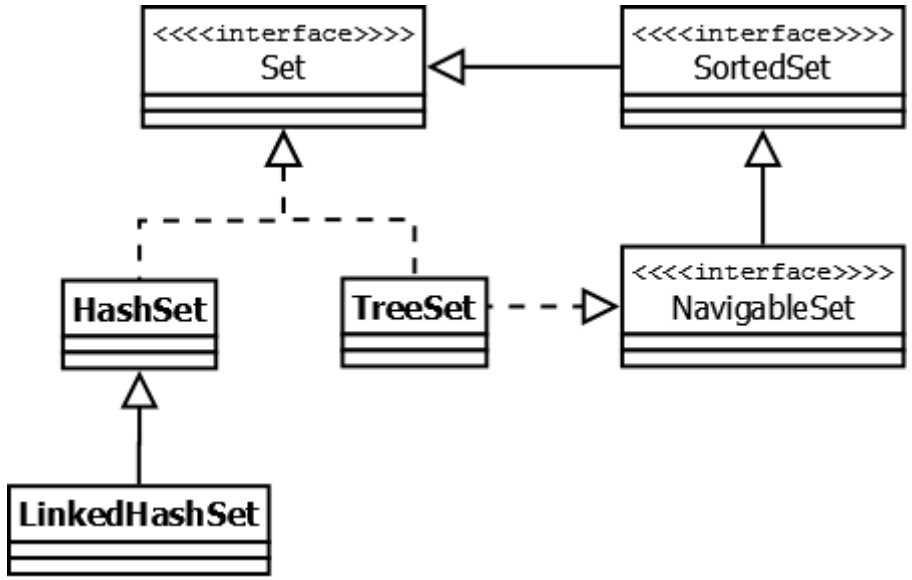
Remove element:

```
prQueue.remove();
prQueue.poll();
```

SET interface

java.util.Set - an **un**ordered set of elements.

- All elements are unique, duplication items are NOT allowed
- Can store at most one NULL



HashSet

HashSet uses a hash table for storage

- Contains unique elements only
- Does not guarantee the order
- Allows the NULL element
- Constant time performance for the basic operations (add, remove, contains and size)
- User defined classes must ***override*** the ***hashCode***() and ***equals***() methods
- **Non-synchronized**

Create object:

```
Set<Long> approversIds = new HashSet<Long>();
```

Add element:

```
list.add("0"); //adds only if it's not there
list.add(5, "100");
```

Remove element:

```
list.remove("0");
list.remove(5);
```

LinkedHashSet

LinkedHashSet - hash table and **LinkedList** implementation of the Set interface

- Maintains insertion order
- Permits NULL elements
- Contains unique elements only
- **Non-synchronized**

Create object:

```
Set<Long> linkedHashSet = new LinkedHashSet<Long>();
```

Add element:

```
list.add("0");
list.add(5, "100");
```

Remove element:

```
list.remove("0");
list.remove(5);
```

TreeSet

TreeSet - implements the Set interface that uses a tree for storage

- **Does not** allow NULL elements
- Contains unique elements only
- Maintains ascending order.
- **Non-synchronized**

Create object:

```
Set<String> treeSet = new TreeSet<String>();
```

Add element:

```
list.add("0");
list.add(5, "100");
```

Remove element:

```
list.remove("0");
list.remove(5);
```

Homework

- Iterator - https://www.hackerrank.com/challenges/java-iterator/problem
- Write a Java program to replace the element on even position of a ArrayList with the specified element (element is hardcoded in the code). {1,2,3,4,5,5,7,2,1,3,9} -> {3} -> {1,3,3,3,5,3,7,3,1,3,9}

Books

- Java, Head First
- Thinking in Java, Eckel

Links

- https://docs.oracle.com/javase/tutorial/collections/interfaces/index.html
- https://www.javatpoint.com/collections-in-java
- https://www.geeksforgeeks.org/collections-class-in-java

Next class

- Map
- Algorithms

Thanks