Collections in Java

queue linked list 等数据结构都是

Collections Overview

- Data structure implementations found within the `java.util` package
- Nicely organized into a class hierarchy of: interface -> abstract class -> (multiple) concrete classes
- We'll focus primarily on List / Set / Map (interfaces) and the most commonly used concrete implementations ArrayList / HashSet / HashMap

Collection interface

- The Collection interface is the super interface/type of almost all the concrete types within the Java collections classes.
- Generically defined
  - `public interface Collection<E>`
- Defines generic method useful for all implementations of a collection. The most commonly used are:
  - boolean contains(Object obj)
  - boolean add(E element)
  - boolean remove(Object obj)
  - void clear()
  - Iterator<E> iterator()

Collection Usage

```
1  public class CollectionExample {
2
3      public static void main(String[] args) {
4          Collection<String> collection = CollectionsFactory.getCollection("foo", "bar");
5
6          collection.add("something else");
7          int size = collection.size(); // 3
8          boolean notTrue = collection.isEmpty(); // false
9          collection.remove("something else");
10         size = collection.size(); // 2
11         collection.clear();
12         size = collection.size(); // 0
13         collection.isEmpty(); // true
14     }
15
16 }
```

Iterator

- Defines a way to traverse elements of a Collection

```
1  public static void main(String[] args) {
2      Collection<String> collection = CollectionsFactory.getCollection("foo", "bar");
3
4      Iterator<String> iterator = collection.iterator();
5      while (iterator.hasNext()) {
6          String next = iterator.next();
7          // do something interesting
8      }
9  }
```

Iterator / for-each

- Anything that implements interface Iterable<T> can be used in a for-each; which includes Collection types.

```
38  public interface Iterable<T> {
39
40      /**
41       * Returns an iterator over a set of elements of type T.
42       *
43       * @return an Iterator.
44       */
45      Iterator<T> iterator();
46  }
47
```

只要collection implement了Iterable

就可以用 for (:) 去

loop

相当于自动多了上面4-6行

Iterator / for-each (cont)

```
1  public static void main(String[] args) {
2      Collection<String> collection = CollectionsFactory.getCollection("foo", "bar");
3
4      for (String value : collection) {
5          // do something interesting
6      }
7  }
```

Removal

- What does this program print?

```
1  public static void main(String[] args) {
2      Collection<String> collection = CollectionsFactory.getCollection("foo", "bar");
3      System.out.printf("%d%n", collection.size());
4      for (String value : collection) {
5          System.out.printf("%s%n", collection.remove(value));
6      }
7      System.out.printf("%d%n", collection.size());
8  }
```

2

true：移除remove了东西

Removal (cont)

- Now what does it print?

```
1  public static void main(String[] args) {
2      Collection<String> collection = CollectionsFactory.getCollection("foo", "bar", "third");
3      System.out.printf("%d%n", collection.size());
4      for (String value : collection) {
5          System.out.printf("%s%n", collection.remove(value));
6      }
7      System.out.printf("%d%n", collection.size());
8  }
```

3

true

ConcurrentModificationException

- WAT?!?!

Removal (cont)

- Before explaining what went wrong (exactly) here's how to correctly remove from a Collection while iterating.

不允许用for each写
要用iterator.hasNext来写

```
1  public static void main(String[] args) {
2      Collection<String> collection = CollectionsFactory.getCollection("foo", "bar", "third");
3      System.out.printf("%d%n", collection.size());
4      Iterator<String> iterator = collection.iterator();
5      while (iterator.hasNext()) {
6          iterator.next();
7          iterator.remove();
8      }
9      System.out.printf("%d%n", collection.size());
10 }
```

Removal (cont)

- Only safe way to remove from a Collection while iterating is to use the Iterator `remove` method.
  - Careful, must call `next` method before calling `remove`.
- So what went wrong? As always, look at the code…
  - Generated for-each uses the Iterator internally (as we have done with a while loop and a `hasNext` call).
  - The `hasNext` checks the cursor size against the collection size
  - The next call checks for modification.
  - SO -> with only 2 elements in the list the iteration is short-circuited before the `ConcurrentModificationException` can be thrown.

bug

List Interface

- The List interface extends Collection
- Used when order matters (think of List as a replacement for arrays).
- Two main concrete types of List
  - ArrayList
  - LinkedList
- Use ArrayList when **random access** is important. It is also dynamically resized. Internally it is implemented with an array that is resized (doubling each time resizing is necessary)
- Use LinkedList when **fast insertions** is important. Internally it is implemented as a doubly linked list.
- There's also something called **Vector**. Do not use this class. It is synchronized but as we'll see in the Concurrency lecture there are much better ways now to handle concurrency.

List (cont)

```
1  public static void main(String[] args) {
2      List<String> list = new ArrayList<String>(2);
3      list.add("foo");
4      list.add("bar");
5      System.out.printf("%dnd element is %s", 2, list.get(1));
6  }
```

- Note the argument to the constructor - expected size of the List. Always do this. Otherwise you'll have lots of resizing operations.

Set Interface

- The Set interface extends Collection
- Used when fast retrieval at an unknown location is necessary.
- Two main concrete types of Set
  - HashSet
  - TreeSet
- Use **HashSet** as it has constant time performance for most common operations add/remove/contains/size. It, however, gives no guarantee as to the iteration order of elements.
- Use **TreeSet** when iteration order is important and needs to be sorted. Order is given at a performance cost for operations add/remove/contains - log(n)

Set (cont)

```
1  public static void main(String[] args) {
2      Set<String> set = new HashSet<>(2, 1.0f);
3      set.add("foo");
4      set.add("bar");
5      System.out.printf("Set contains foo? %s%n", set.contains("foo"));
6  }
```

- Note the argument to the constructor - expected size of the Set. Do this when possible. Otherwise you'll have lots of resizing operations. The second argument is the "load factor"; i.e., at what point should the set be resized.

HashSet Gotcha

```java
public class Item {

    private String value;

    public Item(String value) {
        this.value = value;
    }

    public String getValue() {
        return value;
    }

    public void setValue(String value) {
        this.value = value;
    }

    @Override public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (o == null || getClass() != o.getClass()) {
            return false;
        }

        Item item = (Item) o;
        return (value == null ? item.value == null : value.equals(item.value));
    }

    @Override public int hashCode() {
        return value != null ? value.hashCode() : 0;
    }
}
```

HashSet Gotcha (cont)

- What does this print?

```java
public static void main(String[] args) {
    Set<Item> set = new HashSet<>(2, 1.0f);
    Item foo = new Item("foo");
    set.add(foo);

    System.out.printf("%s%n", set.contains(foo));
    System.out.printf("%s%n", set.contains(new Item("foo")));

    foo.setValue("foos");

    System.out.printf("%s%n", set.contains(foo));

}
```

Interview problem

true

根据equals定义 true

Immutable variable is better!

不会改变它hash的结果,故set不更新

此句不好 改变值最好新建object

“foo”被hash到了1个地方. hashset会去查看那个地方来确认“foo”在不在,那个地方用其唯一的hashCode表示,可能指向linked list的某个node

false (当foos和foo的hashcode恰好一样时,会true)

Map Interface

- Does **not** extend Collection.
- Very often used. Maps a key to a value.
- Three common types.
  - **HashMap** - most common. Uses a hash table implementation.
  - **TreeMap** - analogous to TreeSet, used for ordering. Uses a red-black tree data structure.
  - **LinkedHashMap** - hybrid - linked-list and hash-table implementation. Uses a doubly linked list to maintain ordering of keys.
- There's also a type called **Hashtable** but do **not** use. Other implementations are better. Similar to **Vector** this is synchronized but with Java 1.5 there are many better alternatives (which we'll discuss in the Concurrency lectures).

Map (cont)

multimap key → list of value

```
1  public static void main(String[] args) {
2      Map<String, Integer> map = new HashMap<>(2, 1.0f);
3      map.put("foo", 12);
4      map.put("bar", 10000);
5
6      System.out.printf("Key foo is mapped to %d%n", map.get("foo"));
7  }
```

No add method! not extended from Collection.

get 方法 key 返回 value

key

- Note the argument to the constructor - expected size of the Map. Do this when possible. Otherwise you'll have lots of resizing operations. The second argument is the "load factor"; i.e., at what point should the map be resized.

Additional Data Structures

- **Queue** - push/pop operations. Many implementations for FIFO or LIFO
  - **Deque** - double ended queue
- **PriorityQueue** - uses a heap data structure to maintain first element based on the priority ordering. Uses a **Comparator** to maintain order.
- **NavigableSet** - sorted **Set** with methods to return elements of closest match; i.e., floor or ceiling to some element or return a collection of elements 'higher' than another.
- There's also many **Concurrent** versions (i.e., **ConcurrentHashMap**) which offer performant implementations within a threaded context. We'll talk about these in the Concurrency lectures.

compare 返回integer，表示-2种关系
→ 控制 priority, order

comparable

Convenience methods/constructors

- There are many methods defined within Collection / Map which aid in batch operations and constructor of a collection from another.

```
1  public static void main(String[] args) {
2
3      Collection<String> collection = CollectionsFactory.getCollection("foo", "bar");
4
5      List<String> list = new ArrayList<>(collection);
6
7      Set<String> set = new HashSet<>(2);
8      set.addAll(list);
9
10     collection.removeAll(list);
11
12     set.retainAll(list);
13
14 }
```

collection level 相当于依次调用 add

删掉都删掉 只保留两个中的元素

Views / Wrappers

- Convert an array to a List

```
1  // Arrays.asList takes a varargs
2  List<String> list = Arrays.asList("foo", "bar", "something else");
```

- Get a sub-list from a List

```
1  // careful! bounds check may result in an ArrayOutOfBoundsException
2  List<String> partial = list.subList(1, 10);
```

Views / Wrappers (cont)

- `Collections.unmodifiableCollection(Collection)`
- `Collections.unmodifiableSet(Set)`
- `Collections.unmodifiableList(List)`
- `Collections.unmodifiableMap(Map)`
  - No modifications allowed. However, if this is the desired behavior use Guava (library)'s immutable collections (known immutable improves performance too - again immutable things are great!)
- `Collections.synchronizedCollection(Collection)`
- `Collections.synchronizedSet(Set)`
- `Collections.synchronizedList(List)`
- `Collections.synchronizedMap(Map)`
  - Synchronizes the object. However, do not use. Much better implementations now available in concurrent package.
- Collections.checkedCollection(Collection, Class)
  - Returns a runtime safe collection. I.e., parameters are checked for type correctness at runtime. Not often necessary, most common would be legacy code. Instead ensure you correctly leverage generics and type safety from compiler

Algorithms

- `Collections` class defines many commonly used algorithms.
  - `Collections.sort` - variant of merge-sort. Sorts based on `Comparator`.
  - `Collections.reverse` - useful when necessary. Don't write your own.
  - `Collections.shuffle` - useful when necessary. Don't write your own.
  - `Collections.binarySearch` - extremely useful. Prefer when searching for an item within a collection that doesn't offer O(1) lookup.
  - `System.arrayCopy` - use this to copy array types. Much more performant than manually copying (iterating and copying) as it uses native code.

Read Chapter 14

All sections

Homework 8

https://github.com/NYU-CS9053/Spring-2016/homework/week8