Lecture 2 & Lab 2

2.1 Generics(since JDK 5.0)

Advantages:

1. No need for type cast(type-safety)

Terms:

```
List<E>; //generic type
E; //Formal type parameter
List<String>; //Parameterized type
String; //Actual type parameter
List;   //raw type(原生类型，在JDK5.0前使用)
// we should avoid using raw types
```

Using Generics

1. Generics classes

```
public class gen<T>{
    ...
    //we can use type T or any data structure with type T here
}
```

2. Generics interfaces, for example, Comparable<T>

```
public interface Comparable<T>
{
    int compareTo<T> (T o);
}
```

then we should override the method to use it

```
public myInt implements Comparable<myInt>, Comparable<myInt>{
    //实现了对不同类型的比较
    int data;
    public int compareTo<myInt>(myInt m){
        if(data == m.data){
            return 0;
        }else if(data > m.data){
            return 1;
        }else {
            return -1;
        }
    }
    public int compareTo<int>(int i){
        if(data == i){
            return 0;
        }else if(data > i){
            return 1;
        }else {
            return -1;
        }
    }
}
```

3. Generics methods

```
public static <E> Set<E> union(Set<E> s1, Set<E> s2){
    //the type parameter should be declared right after the static
    Set<E> result = new HashSet<>(s1); //type in <> can be omitted
    result.addAll(s2);
    return result;
}
```

Bounds for Type Varibles

1. form

```
<T extends superclass & interface1 & interface2>;
//use & to seperate classed and interfaces
//superclass should be the first one
```

2. application

```
public static<T extend Comparable> Pair<T> minmax(T[] a){
    ...
}
```

Wildcards

1. create a relationship between generic types

```
String is a subclass of Object
List<String> is not a subclass of List<Object>
```

for parameter-matching in the methods, classes, or interfaces

```
public staic void process(List<?> list){
    //we can pass List<String> to this method
}
```

2. set bounds for wildcards

```
//"extends" can be either direct or indirect
List<? extends superclass>;
//we can pass List<T>, where T extends superclass
List<? super subclass>;
//we can pass List<T>, where subclass extends T
```

Type Erasure

T is converted to Object.

no generics at all in JVM.

2.2 Abstract Data Type(ADT)

Primitive types:

1. **values** immediataly map to machine representations
2. **operations** immediataly map to machine representations

ADT:

1. A type for objects whose behavior is defined by a set of values and a set of operations
2. **Hide** how values are stored in memory and operations are implemented
3. clients only know the data options which can be accessed.

Operations of ADT

1. creater: create new objects of the types
2. producers: create new objects from old objects
3. observer: return a different type for the current ADT
4. Mutators: modify objects itself

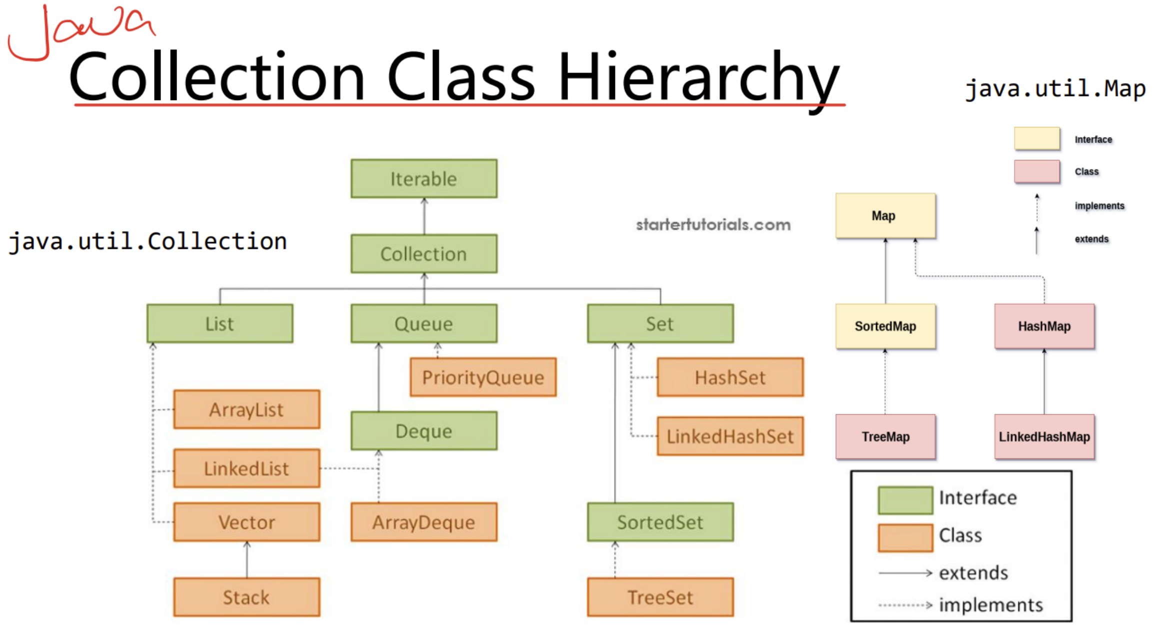
2.3.1 Collections

1. a group of objects
2. mainly used for data storage, data retrieval(检索), and data manipulation

2.3.2 Java Collections Framework

Interfaces

1. Collections Class Hierarchy



2. **Iterable<T>** interface:

```
//impletmenting this interface allows an object to be the target of the "foreach" statements
public interface Iterable<T> //可迭代的，即一定会有一个迭代器
{
    Iterator<T> iterator();
}
```

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();
    void remove(); //删除目前迭代器迭代到的元素
    //注意不能连续使用两次remove，因为迭代器所迭代到的对象已经被删除
    //解决方法1：remove之后要调用iterator.next()
    //解决方法2：使用foreach,每迭代一次会自动调用迭代器方法
}
```

3. **Collection** Interface

```
public interface Collection<E>{
    int size;
    boolean isEmpty();
    boolean contains(Object element);
    //增加，删除元素不强制重写
    boolean add(E element);//optional
    boolean remove(Object element);//optional

    //从一个假头开始
    Iterator<E> iterator(); //返回该对象的迭代器

    Object[] toArray(); //返回一个包含集合中所有元素的 Object 类型数组
    /*
    1.方法的参数 a 是用于指定数组类型和大小的数组。
    2.如果 a 数组的长度大于等于集合的大小，那么集合中的元素将被存储在 a 数组中并返回，
    3.否则将返回一个新的类型为 T 的数组，其中包含集合中的元素。
    */
    E[] toArray(E a[]);

    //Bulk Operations
    boolean containsAll(Collection<?> c); //判断是否包含
    boolean addAll(Collection<? extends E> c); //增加一系列的而元素
    boolean removeAll(Collection<?> c);//删除当前集合与另一个集合c中相同的元素。

    /*保留集合中与另一个集合 c 相同的元素，
    而删除集合中不在另一个集合 c 中的元素。*/
    boolean retainAll(Collection<?> c);

    void clear();//清空当前集合
}
```

4. **Set** interface

    1. Add no methods to Collection
    2. Add stipulation(规定)：no dulplicated elements
    3. redefine some methods of collections<E> or objects
    4. Set idioms

    ```
    //for 2 sets s1, s2
    s1.equals(s2); //比较每个元素地址/hashcode是否相等
    s1.hashcode(); //返回每个元素的hashcode之和,这也是元素的比较核心
    s1.containsAll(s2); //判断s2是否含于s1
    s1.addAll(s2); //取并集并存到s1中
    s1.retainAll(s2); //取交集并赋值给s1
    s1.removeAll(s2); //s2 - s1
    ```

5. **List** interface

    1. List have **"order"**
    2. Add methods

    ```
    int indexOf(Object o); //返回列表中o相同的第一个对象
    int lastIndexOf(Object o); //返回列表中o相同的最后一个对象
    List<E> subList(int from, int to);// 返回[from, to)的子列表
    ```

6. **Map** interface

```
public interface Map<K, V>{
    int size();
    boolean isEmpty();
```

```
    boolean containsKey(Object key); //判断是否包含一个键
    boolean containsValue(Object value); //判断是否包含值
    V get(Object key); //通过键取值
    V put(K key, V value); //存放键值对, optional
    V remove(Object key); //删除键值对并返回值，optional
    void putAll(Map<? extends K, ? extends V> t);//optional
    void clear();

    //from a collection view

    public Set<K> keySet();
    public Collection<V> values();

    //返回所有键值对组成的集合
    public Set<Map.Entry<K, V>> entrySet();
}
```

7. Overriding **equals()**

   List, Set, Map都具有equals()方法来比较两个对象是否相等，首先都是比较大小。随后，

   List与Set都是每个元素进行equals比较，最后全部结果取交集

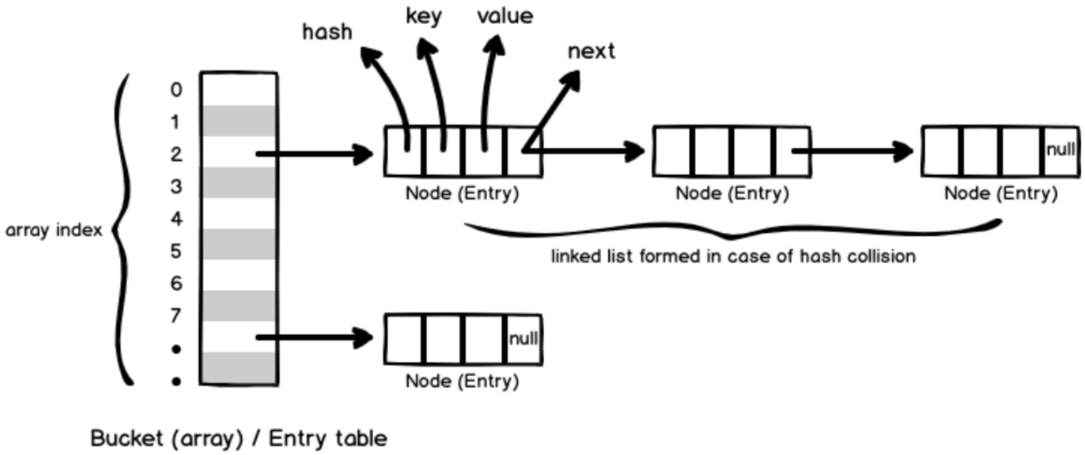
   Map则是取出每个键值对，看看是否存在于第二个map中（这里分别使用到键的equals和值的equals），最后全部结果取交集。

   (by chatgpt)

Implementations

1. **List** Implementation
   1. ArrayList interally uses an array to store the elements.
      remark: ArrayList 底层工作原理
      1. 当创建一个 `ArrayList` 对象时，会创建一个数组来存储元素，这个数组的**默认容量为 10**。当向一个已满的 `ArrayList` 中添加新的元素时，`ArrayList` 会创建一个新的数组，并将原来的元素**复制**到新数组中，然后将**新元素添加到新数组**中。
      2. 在扩容时，一般情况下增长因子的值为 **1.5。这个过程比较耗费时间和内存，因此，在创建 `ArrayList` 对象时，可以通过指定初始容量来减少扩容的次数，从而提高效率。**
      3. **如果需要频繁地进行插入、删除等操作，可以考虑使用链表实现的 `LinkedList` 类。**
2. **Map** Implementation

1. HashMap struture



HashMap

2. Map Implementation -- HashMap

```
step1: map.put(key, value);
step2: 计算key.hashcode();
step3: 通过hashcode,计算出哈希桶的下标;
step4: 判断是否发生哈希碰撞
    没有发生哈希碰撞:
        在哈希桶的对应位置链接，成为第一个节点
    发生哈希碰撞：判断key.equals(existing_key)
        相等：替换当前节点
        不相等：链接当前节点，成为下一个节点
```

```
//看看有什么不同
//FB的哈希值为2236，LD的哈希值为2236，Ea的哈希值为2236
map.put("FB", 1);
map.put("LD", 2);
map.put("Ea", 3);
System.out.println(map);
map.put("FB", 4);
System.out.println(map);
```

3. hashCode() and equals()

   1. `hashCode()` convert interal address to an int and return
   2. `equals(Objected)` compared hashcode by default.
   3. `==` always compared hashcode

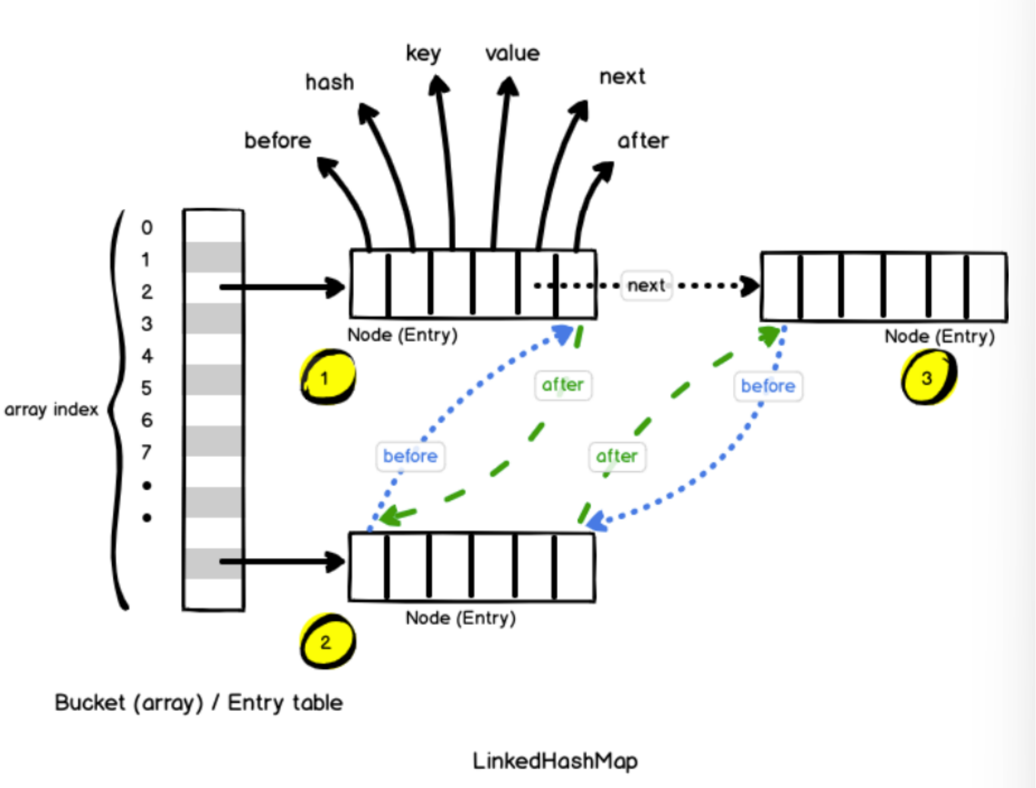
4. Map Implementation -- LinkedHashMap

   1. difference from HashMap:

      adding two pointer: before and after

      aim: preserve the insertion order of keys (we can access the element by insertion order too)

2. structure graph



3. Remark:

   When LinkedHashMap maintains the mapping relationship, it needs to use **additional memory** to store the **order** relationship between key-value pairs. Therefore, when making a trade-off between space and time, an appropriate data structure should be chosen to meet the needs. (by chatgpt)

5. Map Implementation -- TreeMap

   1. charecteristics:

      1. when **keys need to be ordered** using natural ordering or by a comparator

      2. keep the relative size relationship of keys

      3. underlying structure **Red-Black Tree**

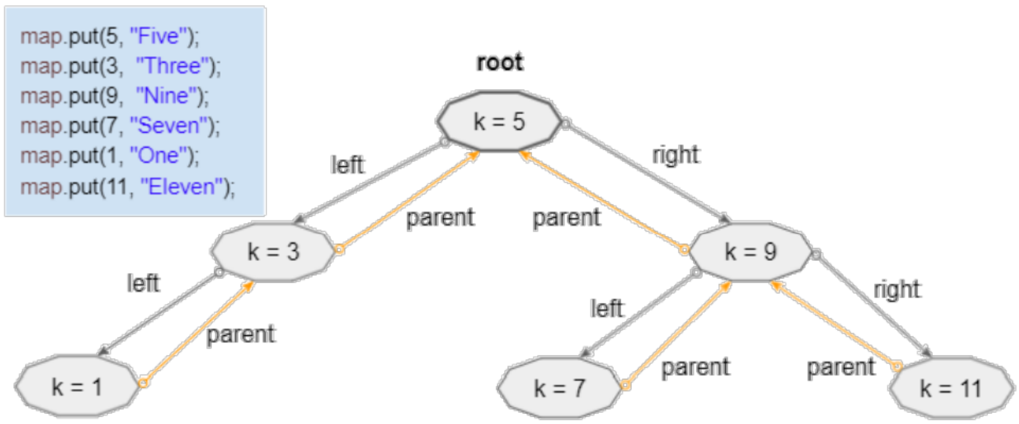
   2. some interfaces

```
        put(K key, V value); //将指定的键值对插入到 TreeMap 中。O(logn)
        remove(Object key); //从 TreeMap 中删除具有指定键的元素。O(logn)
        get(Object key); //返回具有指定键的值，如果不存在，则返回 null。O(logn)

        firstKey(); //返回 TreeMap 中的第一个（最小的）键。O(logn)
        lastKey(); //返回 TreeMap 中的最后一个（最大的）键。O(logn)

        higherEntry(K key); //返回一个大于k的最小键所在的键值对(Map.Entry), O(logn)
        lowerEntry(K key); //返回一个小于k的最小键所在的键值对(Map.Entry), O(logn)
        floorEntry(K key); //返回一个小于等于于k的最小键所在的键值对(Map.Entry), O(logn)
        ceilingEntry(K key); //返回一个大于等于于k的最小键所在的键值对(Map.Entry), O(logn)
```

3. structure



```
map.put(5, "Five");
map.put(3,  "Three");
map.put(9,  "Nine");
map.put(7, "Seven");
map.put(1, "One");
map.put(11, "Eleven");
```

Source: https://o7planning.org/13597/java-treemap

3. **Set** Implementation

	HashSet	**LinkedHashSet**	**TreeSet**
Base	HashMap	LinkedHashMap	TreeMap
Property	doesn't maintain anything	maintain insertion order	maintain sorting order
Performance Ranking	1	2	3
NULL Tolerance	Maximum one	Maximum one	doesn't allow
Printing	unordered	by insertion order	by sorting order

4. Common Inplemetation Behaviours
    1. all implement tation permit **null** elements, key, and , values(Except TreeSet, TreeMap)
    2. serializable: **Serialization** is also an important feature for many Java applications, as it allows objects to be saved to disk or transmitted over a network. (by chatgpt)
    3. **not thread-safe** by defaut
    4. all have fail-fast iterator:

       detecting illegal **concurrent modification** and throw an exception.

Algorithms

1. difference between **Collections** and **Collection<E>**

   Collections is a class that provides utility methods for **working with collections**, whereas **Collection<E>** is an interface that defines the basic operations that all collection classes should support. **In short, that is commonality and individuality.**

2. reusable(generic) algorithms

   static methods in the **collections**(not collection)

```
public class Collections extend Object{
    static <T extends Comparable<? super T>> void sort(List<T> list);
    static int binarySearch(List list, Object key);
    static <T extend Comparable<? super T>> T min(Collection<T> coll);
    static <T extend Comparable<? super T>> T max(Collection<T> coll);
    // fill a same elements
    static <E> void fill (List<E> list, E e);
    static <E> void copy(List<E> dist, List<? extends E> src);
    static void reverse(List<?> list);
    static void shuffle(List<?> list);
}
```

3. why reusable algorithms

   No need to write, test, debug in different types

4. **sorting** algorithm: reorder a collection according to **natual ordering**
    1. **String** and **Date** implement the **compareTo(T o)** allowing objects to be sorted automatically
        1. **File**: system-dependent **lexicographic**
        2. **String**: **lexicographic**
        3. **Date**: **Chronological**
    2. **boolean** false < true
    3. Collections.sort(list) will throw a **ClassCastException** if T do not implement Comparable
    4. **Comparable** and **Comparator**

       Comparable interface is used to define their natural ordering,

       Comparator object is used to define a order by different properties
    5. Sorting by Comparable and Comparator

1. Collections.sort() overview

   by **Comparable**

```
public static <T extends Comparable<? super T>> void sort(List<T>
list)
```

   by **Comparator**

```
public static <T> void sort(List<T> list, Comparator<? super T>
comparator)
```

2. T extends Comparable<? super T>

   we are madatory to override a method

```
// this method define a order to be sorted
public int compareTo(T o);
```

3. Comparator<? super T> comparator

   Actually, there is no instantiation of the Comparator. So we should define **at least** a class which implements **Comparator<? super T>**

```
// we define class
class order1<T>{
    //madatory to override
    public int compare(T o1, T o2){
        // return a negative number if o1 < o2
        // return 0 if o1.equals(o2)
        // return a positive number if o1 < o2
    }
}
class order2<T>{
    //madatory to override
    public int compare(T o1, T o2){
        // return a negative number if o1 < o2
        // return 0 if o1.equals(o2)
        // return a positive number if o1 < o2
    }
}
```

4. usage

```
Collections.sort(list);
Collections.sort(list, new order1());
Collections.sort(list, new order2());
```

Convenience Operations

1. **Arrays.asList(T... a)**

    1. transfer an array to a list
    2. served as a bridge between array-based and collection-bases API
    3. we don't need to create an `ArrayList<T>` and insert all the array elements

2. **Collections.nCopies(int n, T o)**

    1. returns an **immutable list** consisting of n copies of the object o
    2. Useful in combination with the `List.addAll()` method to grow lists with **duplicated elements**

    ```
    List<Type> list = new ArrayList<Type>(Collections.nCopies(1000, (Type)null));
    pets.addAll(Collections.nCopies(3, “cat”));
    ```

3. **Collections.singleton(T o)**

    1. return an **immutable set** containing only the specified object o
    2. useful in conbination with the `list.removeAll` method.

    ```
    List<String> list = Array.asList({"C++", "Java", "C++"});
    list.removeAll("C++");
    System.out.println(list);
    ```

4. **empty**

    1. sometimes we need an **empty set/map/list** as an **argument** or **return value**
    2. when using `emptySet(), emptyMap(), emptylist()` repeatedly, actually it return a **same, immutable** collection, which greatly **reduces the cost of memory** and can be used **without fearing modified**

    ```
    Set<String> set1 = new HashSet<String>();
    Set<String> set2 = new HashSet<String>();
    Set<String> set3 = Collections.emptySet();
    System.out.println(set1 == set2); // false
    System.out.println(set1 == set3); // false
    System.out.println(set2 == set3); // false

    Set<String> set4 = Collections.emptySet();
    Set<String> set5 = Collections.emptySet();
    System.out.println(set3 == set4); // true
    System.out.println(set4 == set5); // true
    ```

Suggestions to Grasp the Functions of Collections

1. by Further Reading: Official Website

   Reference:

   > Trail: Collections: Table of Contents (The Java™ Tutorials) (oracle.com)

2. In IDE

   1. look up all the prompt words
   2. `ctrl + left click` on some functions you want to know deeper