第一部分：类集

类集是JDK1.2版本提出，主要作用是解决数组长度固定的问题。类集其实就是类似于数据结构，是对数据更高效率的一种存储。

类集主要有Collection接口和Map接口。

Collection接口中常用方法为add()、iterator()方法，分别为向集合中添加元素和将集合元素迭代输出。但Collection接口无法区分存放数据类型(是否重复)，故一般我们都不直接使用Collection接口，而是使用其下的List接口(元素允许重复)和Set接口(元素不允许重复)。

注意：List接口对Collection接口进行扩展，扩展了get()方法(根据下标获取元素),set()方法(set方法两个参数，前者为下标，后者为插入元素。也就是说set方法为向指定下标写入数据)。但Set接口并没有对Collection接口进行扩展，即Set子类不可使用get方法。

List接口下常用子类为ArrayList(底层数组)、Vector(底层数组)、LinkedList(底层双向链表，默认尾插)

```
//动态数组核心代码
public void grow(){
 int oldCap=elementData.length;
 int newCap=oldCap+(oldCap<64 ? oldCap + oldCap>>1);
 if(newCap > Integer.MAX_VALUE - 8){
 throw new Exception("数组转换异常");
 }
 elementData=Arrays.CopyOf(elementData,newCap);
}
```

问题一：ArrayList、Vector和LinkedList有什么区别?

首先，三者的共同点为均是List接口的子类。

其次它们各自的特点为：ArrayList是JDK1.2提出，底层由数组实现，线程不安全，异步处理性能较高；Vector是JDK1.0提出，底层由数组实现，线程不安全，同步处理性能低；LinkedList由JDK1.2提出，底层由双向链表实现，线程不安全，异步处理性能较高。

最后，三者的区别主要有以下五个方面：

ArrayList与Vector比较：

第一：出现版本不同。ArrayList由JDK1.2提出，Vector由JDK1.0提出。

第二：支持遍历方式不同。Vector支持枚举输出，而ArrayList不支持。

通过简单阅读源码可以得到以下三个区别：

第三：初始化策略不同。ArrayList采用懒加载策略，在元素第一次添加时才将数组长度开辟为默认初始长度10或自定义初始长度，在元素第一次添加之前数组长度为0；而Vector在类加载时直接将数组长度初始化为10。

第四：扩容机制不同。ArrayList每次扩容为原先的1.5倍，而Vector每次扩容为原先的2倍。

第五：安全机制不同。ArrayList线程不安全，采用异步处理性能较高；Vector线程安全，采用内建锁，同步处理性能较低。

ArrayList与LinkedList比较：

ArrayList底层依据数组实现，遍历时间复杂度为O(1)；LinkedList底层依据双向链表实现，遍历时间复杂度为O(n)。

ArrayList为List接口最常用子类。

集合中存放最多的一般是简单Java类。

那么当集合中存放的是Java类时，需注意使用remove()、contains()方法时一定要在简单Java类中覆写equals()方法。原因是若集合中存放的是匿名对象，因没有名称就无法直接查找、删除，需借助equals()方法判断该匿名对象是否存在。

```
1 public boolean equals(Object o){
2  if(this==o) return true;
3  if(o==null || !(o instanceof Person)) return false;
```

```
    Person p=(Person) o;
    return p.name.equals(o.name) && p.age.equals(o.age);
}
```

ArrayList适用大量存取和删除操作。
LinkedList更适用于更多进行插入删除操作多且不会随机访问数据的代码。

Set接口
Set接口下常用子类为HashSet(无序存储)--->数据输出顺序不一定是存储顺序
TreeSet(有序存储) 序指的是自定义类直接实现内部比较器(Comparable接口)时就表明该类中元素具备排序能力，具体以compareTo()方法中的定义为主；或者自定义类可传入外部比较器(Compartor接口)，序依据外部比较器中compare()方法中定义的为主。
注意：若Java简单类实现了Comparable接口，就表明该类支持排序，存放该类的Collection或数组可直接通过Collections.Sort()或Arrays.Sort()进行排序。
该类可直接存放在TreeSet或TreeMap中。
Set常用子类：
1.HashSet
底层使用哈希表+红黑树
允许存放null，无序存储
2.TreeSet
底层使用红黑树
不允许存放null，有序存储
(若向TreeSet中存空值，则编译正常运行时报空指针异常)

若要将自定义类存储到TreeSet中，则该类必须实现Comparable接口或传入Comparator对象。

Set集合为什么不允许数据重复?

原因是存入Set集合的元素实际上是存储至Map集合中的key值处，且value值相同，即Set本质上是一组value值相同的Map集合。

Comparator接口中有两个抽象方法为compare()和equals()，为什么外部比较器类继承该接口时只需要覆写compare()方法而不需要覆写equals()方法呢?

原因是在Java中所有类均默认继承Object类，Object类已经覆写了equals()方法，普通方法继承了Object类覆写的equals()方法。

TreeSet使用升序排序实现。

TreeSet和TreeMap判断集合中元素是否重复时是根据compare()方法或compareTo()方法中定义，一般若自定义类中存在多个属性时，排序方法中也应当对所有属性进行大小比较，否则若比较属性元素有所遗漏，那么当有两个元素排序方法中所涉及到的所有属性相同而其他属性内容不同时，此时会判断为这两个元素相同，即元素已存在，出现偏差。

HahsCode和equals在HashSet和HashMap判断重复元素时同时覆写。

判断重复元素的主要步骤为先使用hashCode获得该元素的hash码，找到对应桶再使用equals()方法判断该桶中是否已经存在要插入元素。若不存在则尾部插入，若存在则忽略。

```
1 public int hashCode(){
2  return Objects.hash(name,age);
3 }
```

HashSet和HashMap中是如何存放数据的?

首先调用hashCode计算出当前元素的hash值，找到对应的桶，在使用equals()方法判断该桶中是否已经存在该元素，若不存在则尾插入队；若存在则忽略。

为什么要同时覆写hashCode和equals()方法?

原因是hashSet和hashMap进行重复元素判断时需首先通过hashCode方法得到该元素的hash码找到对应桶，再使用equals()方法判断该桶中是否已存在该元素，若是则不再添加元素，若不是则尾插进链表即可。即两个对象的equals()方法返回true则表明hashCode必然相等，但hashCode相等equals不一定相等，只有当且仅当hashCode与equals均相等才认为该元素已存在。

集合输出：Iterator、ListIterator、Enumeration、foreach。

注意：Iterator只有Collection接口中定义，也就是说只有其子类才可使用，Map接口中并没有对其定义，故Map接口子类若要使用集合输出就需要先将Map转换为Set，再进行输出。

Iterator(迭代输出)--->只能从前向后输出(collection接口提供)

```
Set<Person> set=new HahsSet<>();
Iterator iterator=set.iterator();
while(iterator.hasNext()){
 System.out.println(iterator.next());
}
```

Iterator是接口，remove为删除元素的方法，该方法从JDK1.8变为default完整方法。remove方法主要是解决了集合内容的删除操作。但需值得注意的是在输出过程中，若使用集合提供的remove()方法进行与元素删除则会报并发修改异常，而使用迭代器输出过程中删除则正常删除。原因是在遍历过程中使用集合的remove方法不能保证同步，但使用迭代器中remove()方法则是一次hasNext()一次next()。

单纯遍历使用for/foreach即可，若要边遍历边删除则使用迭代器输出。

那么在遍历过程中为什么使用集合提供的remove()方法就会抛出异常而使用迭代器的remove()方法就正常呢?

原因是此处存在一个fast-fail(快速失败)策略。在集合的add()、remove()方法中存在一个modCount变量，该变量作用是记录当前集合被修改的次数(此处修改指的是修改集合框架)，同时还存在一个该变量的副本，而将修改次数赋值副本是在集合遍历输出之前，故使用集合输出过程中一旦使用集合的增加修改方法对集合进行修改，此时modCount++,与副本值不相同会抛出并发修改异常。

至于集合输出过程中使用迭代器的remove()方法为什么可以正常删除，原因是在集合输出过程中一旦调用迭代器的remove()方法则会立刻更新副本值，也就是说修改次数与其副本值相同，故可正常删除不会报异常。

快速失败策略保证了所有用户在迭代遍历集合时看到的数据都是最新的，避免了脏读现象。

fail-safe策略就是不会发生并发修改异常，juc包下的所有线程均是安全集合

Enumeration枚举输出---只有Vector及其子类支持

```
List<String> vector=new Vector<>();
Enumeration<String> enumeration=vector.elements();
while(enumeration.hasMoreElements()){
 System.out.println(enumeration.nextElements());
}
```

Enumeration应用场景：使用第三方的库，版本较旧，不支持Iterator时使用Enumeration

foreach输出---所有集合都支持

注意：进行foreach遍历输出的本质在于各个集合内部都内置了迭代器。

Map集合--以键值对形式存放数据最大父接口(key=value,Map最大特点是可通过key找到value)
Collection--将元素单个存储的最大父接口
Map接口常用方法：put(向Map集合中添加元素)、get(根据key值取到相应的value值，若key值不存在返回null)、keySet(返回值类型为Set<K>，原因是该方法为获得所有key值，key值不可重复)、values(返回值类型为Collection<V>,该方法为返回所有的value值，value值可重复)、entrySet(将Map集合转变为Set集合)
Map接口常用子类：HashMap、TreeMap、Hashtable、ConcurrentHashMap
注意：Map集合中key值不可重复，但value值可重复，也就是说一个value值可能对应多个key值。当向集合中添加元素时若key值相同，即变为该key值对应的value值的更新操作，若key值不存在，返回null。

HashMap---Map集合中最常用的子类
HashMap类比于HashSet，判断重复元素时需同时覆写hashCode()方法和equals()方法
HashMap中允许Key值和value值为空，但key值只能有且一个为null，value值可有多个为null。
问题：Map与Set的关系?
Set集合实际上就是Map集合。Set集合中元素存储底层实际上是Map，即Set集合中元素存放至Map集合中的key值，value值全部为空的Object对象。也就是说HashSet就是HashMap,TreeSet就是TreeMap。

HashMap常考方法源码剖析：
默认初始化桶个数为16，负载因子为0.75，树化阈值为8，树化最小元素个数为64，解树化阈值6
树化思想：若桶中元素大于等于8时且同时整张哈希表的元素个数大于64，此时会将桶中元素结构转为红黑树结构；若桶中元素大于等于8但整张哈希表元素总数没有超过64，那么此时只会对该桶进行resize()(扩容为两倍)
树化原因：解决桶中链表太长导致的查找速度过慢的问题，将查找速度由O(n)提高到O(logn)

且HashMap初始化策略同样采用懒加载模式，并不会在对象产生时初始化哈希表，而是在第一次向其添加元素时初始化。

resize()方法源码：
作用：
1.在第一次向哈希表中添加元素时进行哈希表的初始化(初始化桶个数为默认16)
2.当表中元素个数达到阈值(容量*负载因子)后进行扩容为原哈希表的二倍。
3.扩容后，对原来元素进行rehash，要么元素还呆在原桶里，要么呆在double桶里。

put()方法源码思路：
1.首先判断哈希表是否为空，若为空则调用resize()对哈希表进行初始化。
2.对key值做hash，判断是否发生哈希碰撞
    a.若没有发生哈希碰撞，则直接将该元素插入至该桶的头节点
    b.若发生了哈希碰撞

(1)此桶中若已树化，则将该节点构造为树节点插入红黑树

(2)此桶中若未树化，则将该节点尾插入链表中

3.判断该桶中是否与插入节点的key值相等的节点，若有则对该key值的对应的value值做更新操作

4.插入节点后，判断当前链表长度是否大于阈值(容量*负载因子)，若大于则调用resize()方法进行扩容

get()方法源码思路:

1.若哈希表为null或key值为null，返回null；

2.若哈希表不为null且key值不为null，对key值进行hash

a.若key值刚好为该桶的头节点，直接返回

b.遍历桶中其他节点

(1)若桶已树化，调用树的遍历方式找到对应的Node并返回;

(2)若桶未树化，遍历链表找到指定key值对应的Node返回。

hash()方法源码：

对key值做hash，即为(hash ^ (hash>>>16)) & (n-1)

其中n为桶个数(默认为16)

问题一：为什么不采用Object类提供的hashCode方法计算出来的值直接做为桶下标呢?

原因是int型为32位，若直接使用则几乎不会发生哈希碰撞，因哈希表太大，相当于此时哈希表是一个普通数组。

问题二：为什么要hash>>>16?

原因是hash基本上是在高16位进行hash运算，低16位不考虑。

问题三：为什么hashMap中容量均为2^n?

原因是当容量为2^n时，(n-1)&hash就相当于(n-1)%hash,用位运算代替%，提高运算效率。

性能问题：
多线程下，在竞争激烈的场景下使用HashMap会造成CPU100%
解决：使用ConcurrentHashMap代替HashMap
性能主要消耗：在于rehash过程，且开销会越来越大
解决：在能预估存放元素个数的情况下，将哈希表初始化为相应大小，减少扩容rehash的次数
注意：若resize过程中发现桶中红黑树节点小于等于解树化阈值，则会将该桶中红黑树结构退化为链表结构。

Hashtable：单纯的哈希表实现
HashtableJDK1.0提供，线程安全，不允许key值和value值为null
Hashtable在put()、get()、remove()等方法上均使用synchronizd加锁，锁的是当前Hashtable对象，即整张哈希表。也就是说只要有一个线程进入，其他所有线程均需等待(包括其他方法也不可使用)，故Hashtable性能极低。
如何优化性能问题?
JDK7的ConcurrentHashMap思路：
通过对锁的细粒度化，将整张表锁拆分为多个锁进行优化。
两个hash，即哈希表一旦初始化则一共只有16把锁，