1. 映射
2. 散列映射HashMap

散列映射对键进行散列，与键关联的值不进行散列。每次往映射中添加一个对象时，必须同时提供一个键。要想检索一个对象，必须提供一个键调用get方法，如果 映射中没有存储与给定键对应的对象，get方法会返回null。null在很多情况下会引发异常，可以将get方法替换为getOrDefault方法，在没有键对应的值时，返回一个默认值。如下所示：

HashMap<String, String> map = new HashMap<String, String>();  
map.put("1", "ss");  
String orDefault = map.getOrDefault("2", "33");

需要迭代映射中的键和值时，使用forEach方法，提供一个接受键和值的lambda表达式，映射中的每一项会依序调用这个表达式。

map.forEach((k,v) -> System.*out*.println("key = " + k + ", value = " + v));

1. 树映射TreeMap

树映射根据键的顺序将元素组织为一个搜索树，即将键进行了排序。与之前讲过的集Set一样，散列的速度会快一些，如果不需要对键进行排序，选择散列映射HashMap是更好的选择。

1. 更新映射的值

通常情况下，我们可以得到一个与键关联的原值，完成更新，再放回更新后的值。不过，在键第一次出现时，通过get方法得到的就是一个null，当然，我们可以使用getOrDefault方法设置默认值来解决这个问题。

counts.put(word,counts.getOrDefault(word, 0)+1);

或者使用putIfAbsent方法，只有在键原先存在时才会放入一个值：

counts.putIfAbsent(word, 0);

merge方法可以做到更好，如果键原先不存在，将把word和1关联，否则使用Integer::sum函数组合原值和1，如下所示：

counts.merge(word, 1, Integer::sum);

1. 映射视图

我们可以得到映射的视图——实现了Collection接口或某个子接口的对象。映射有三种视图：键集，值集合，键值对集。键和键值对都是集Set，因为键是不能重复的，如下所示：

Set<K> keyset()

Collection<V> values()

Set<Map.Entry<K,V>> entrySet()

keyset()方法得到映射的键集，values()方法得到映射的值集合，entrySet()方法得到映射集，映射集中的元素是实现了Map.Entry接口的类的对象。注意，得到的集或者集合都是不能添加任何元素的，只能通过Map来添加元素！！！

Set<String> strings = map.keySet();  
Collection<String> values = map.values();  
Set<Map.Entry<String, String>> entries = map.entrySet();

如果想同时获取键和值，使用entrySet方法得到映射集，然后对集中的每个键值对对象调用getKey()方法获取键，调用getValue()方法获取值。

for(Map.Entry<String, String> entry : map.entrySet()){  
 String key = entry.getKey();  
 String value = entry.getValue();  
}

1. 弱散列映射WeakHashMap

当一个值所对应的键已经不在程序的任何位置使用时，那么不再有任何途径可以引用这个值的对象了，程序的任何位置不会再有这个键了，无法从映射中删除这个键值对。同时，垃圾回收器也不能删除它，因为垃圾回收器会跟踪活动的对象，只要映射对象时活动的，其中的所有桶也就是活动的，不能被回收，只能由程序来负责从映射表中删除无用的键。

WeakHashMap就是为了解决这个问题而生的，当对键的唯一引用来自散列表映射的值时，WeakHashMap会和垃圾回收器一起删除键值对。怎么做到的呢？WeakHashMap使用弱引用保存键，WeakReference对象将包含另一个对象的引用，在这里，就是一个散列表键。当垃圾回收器发现某个对象只能由WeakReference引用时，垃圾回收器会将其回收，并且将引用这个对象的弱引用放入一个队列中，定期检查这个队列中新添加的弱引用，删除该弱引用对应的键的键值对。

1. 链接散列映射LinkedHashMap

链接散列映射LinkedHashMap会记住插入元素的顺序，这样就避免了元素的随机排列。同时，还可以实现使用访问顺序来迭代处理映射值，每次调用get或者put时，收到影响的项将从当前的位置删除，并放到项链表的尾部（只影响项在链表中的位置，而散列表的桶不会受到影响，映射条目在键散列码对应的桶中）。访问顺序对于实现缓存的“最近最少使用”非常有效，可以根据访问顺序将访问过少的项删除。

var staff = new LinkedHashMap<String, String>();

1. 枚举映射EnumMap

EnumSet是一个枚举类型元素集，枚举类型只有有限个实例，所以EnumSet内部使用位序列实现，如果对应的值在集中，则对应的位为1。EnumSet没有公共构造器，只能通过静态工厂方法构造：

enum WEEKDAY{*MONDAY*, *TUESDAY*, *WEDNESDAY*, *THURSDAY*, *FRIDAY*, *SATURDAY*, *SUNDAY*};  
EnumSet<WEEKDAY> al = EnumSet.*allOf*(WEEKDAY.class);

EnumMap是一个键类型为枚举类型的映射。

1. 标识散列映射IdentityHashMap

该映射的键的散列值不是用hashCode方法计算的，而是用System.identityHashCode方法计算的，这是Object.hashCode根据对象的内存地址计算散列码时使用的方法。且在比较对象时，使用的是“==”，而不是equals，这说明即使两个对象的内容相同，也被视为不同的对象，通常被用于实现对象遍历算法，跟踪哪些对象已经被遍历过。

1. 视图与包装器
2. 视图

之前使用keySet方法获取了映射对象的键集，返回了一个实现Set接口的类对象，由这个类的方法操纵原始的映射，这个返回的集合称为视图。

1. 小集合

Java9引入了一些静态方法，可以生成给定元素的集或列表，以及给定键值对的映射。如下所示：

List<String> names = List.*of*("peter", "paul", "mary");  
Set<Integer> integers = Set.*of*(2, 3, 4);  
Map<String, Integer> socres = Map.*of*("peter", 2, "paul", 3, "mary", 4);

Map接口还有一个ofEntries静态方法，能接受任意多个Map.Entry<K,V>对象，可以使用entry静态方法创建这些对象，如下所示：

Map<String, Integer> scoress = ofEntries(  
 entry("peter", 2),  
 entry("peter", 2),  
 entry("mary", 2)  
);

注意，元素，键或值不能为null。这些集合对象是不可修改的，如果视图修改他们的内容，会抛出一个UnsupportedOperationException异常。如果想要一个可修改的集合，可以讲这个集合对象传到构造器中，如下所示：

var name s= new ArrayList<>(List.*of*("peter", "paul", "mary"));

Collections类中有许多实现方法，如nCopies方法会返回一个实现了List接口的不可变对象，其中包含着n个anObject对象：

Collections.*nCopies*(n, anObject);

1. 子范围

可以为很多集合建立子范围视图，并对子范围应用任何操作，这些操作会自动反映到整个列表中。如下所示，获取列表的10 -19个元素，并删除这些元素：

ArrayList<Integer> nums = new ArrayList<>();  
List<Integer> group = nums.subList(10, 19);  
group.clear();

对于有序集和映射，可以使用排序顺序建立子范围，如下所示，返回大于等于from且小于to的所有元素组成的子集：

SortedSet<E> subset(E from, E to)

1. 不可修改的视图

Collections类中有方法可以生成集合的不可修改视图，这些视图对集合增加了一个运行时检查，如果视图对集合进行修改，则抛出异常。这些方法都是以unmodifiable开头的，每个方法对应一个接口，如Collections.unmodifiableList处理实现了List接口的类。

ArrayList<Integer> nums = new ArrayList<>();  
List<Integer> group = nums.subList(10, 19);  
Collections.*unmodifiableList*(group);

你可以使用这些视图的访问器访问原集合的对象，但是一旦会用更改器方法，就会抛出UnsupportedOperationException异常，而不是将调用传递给底层集合。且视图只是包装了接口而不是具体的集合对象，因此只能访问接口中定义的方法。

注意，unmodifiableCollection方法返回的集合，它的equals方法不会调用底层集合的equals方法，而是继承了Object类的equals方法，这个方法只是检测两个对象是否相等，不会对内容进行检查，hashCode方法也是一样的。而unmodifiableSet和unmodifiableList方法会使用底层的equals和hashCode方法。

1. 同步视图

Collections类的静态方法synchronizedMap可以将任何一个映射转换为有同步访问方法的Map，其他的集合也是一样，只要是以synchronized开头的方法都可以转换。

var map = Collections.synchronizedMap(new HashMap<String, String>());

1. 检查型视图

Collcetions类中的静态方法checkedList能够检查插入的对象是否是正确的类型，如果不是则抛出ClassCastException异常。这种以checked开头的方法创建的集合能够在正确的位置报告错误，如将错误类型的元素混入泛型集合中，能够在插入时就报告这个错误，而不是在调用get方法获取这个元素并进行强制转换时才会出现一个类强制转换异常。

var strings = new ArrayList<String>();

List<String> safeStrings = Collections.checkedList(strings, String.class);

1. 算法
2. 排序sort

Collections类中的sort方法可以对实现了List接口的集合进行排序，这个方法假定列表元素都实现了Comparable接口。

var staff = new LinkedList<String>();

Collections.sort(staff);

如果想按照降序对列表排序，使用静态方法Collections.reverseOrder()返回一个比较器，传入sort函数中：

staff.sort(Collections.reverseOrder());

Java中链表的排序是将所有的元素转入一个数组中，对数组进行排序，然后，再将排序后的序列复制会列表。

1. 二分查找binarySearch

Collections类的binarySearch方法对有序集合进行二分查找，如果集合没有采用Comparable接口的compareTo方法进行排序，还要提供一个比较器对象。

I = Collections.binarySearch(c, element);

I = Collections.binarySearch(c, element, comparator);

如果返回一个非负的值，这表示匹配对象的索引；如果返回一个负值，表示没有匹配的元素，不过可以利用该返回值 i来计算应该将element插入到集合中的哪个位置，以保持集合的有序性。插入的位置应该是 -I – 1。

1. 批操作
2. 成批删除元素

coll1.removeAll(coll2);

从coll1中删除coll2中出现的所有元素。

coll1.retainAll(coll2);

从coll1中删除所有未在coll2中出现的元素。

1. 成批添加元素

coll1.addAll(coll2);

将coll2的所有元素添加到coll1中。

1. 集合与数组的转换
2. 数组转集合

List.of包装器可以实现，如下所示：

String[] values = …

var staff =new HashSet<>(List.of(values));

1. 集合转数组

使用toArray方法得到的只是Object[]对象数组，且不能直接强制类型转换。我们需要为toArray方法提供一个指定类型且指定长度的数组，这样，返回的数组就是指定的类型长度的数组了。

String[] val = new String[2];  
var staff = new HashSet<>(List.*of*(val));  
String[] vals = staff.toArray(new String[staff.size()]);

1. 属性映射

属性映射是一个特殊类型的映射结构，同样是键值对，不过可以保存到文件中以及从文件中加载，且有一个二级表保存默认值。

1. 使用

实现属性映射的Java平台类名为Properties，经常用于程序的配置文件中。

Properties settings = new Properties();  
settings.setProperty("width", "699");

使用store方法将属性映射列表保存到一个文件中，第一个参数是一个输出流，第二个参数是包含在这个文件中的注释：

FileOutputStream fileOutputStream = new FileOutputStream("program.properties");  
settings.store(fileOutputStream, "Settings");

使用load方法从文件中加载属性：

FileInputStream fileInputStream = new FileInputStream("propgram.properties");  
settings.load(fileInputStream);

注意，Properties类实现了Map<Object,Object>，因此可以使用Map接口的get和put方法，不过get方法返回类型为Object，而put方法允许插入任意的对象，因此最好使用处理字符串而不是对象的getProperty()和setProperty()方法。

1. 默认值

为属性映射提供默认值有两种方式，第一种是在每次查找一个字符串的值时指定默认值：

String filename = settings.getProperty(“fileName”, “”);

第二种是将所有的默认值放在一个二级属性映射中，在主属性映射的构造器中传入这个二级映射。

var deSettings = new Properties();  
deSettings.setProperty("width", "600");  
deSettings.setProperty("length", "200");  
var setting = new Properties(deSettings);

1. 位集

Java平台中的BitSet类用于存储一个位序列，由于位集将位包装在字节中，因此使用位集比使用Boolean对象的ArrayList高效地多。

1. get

获取某个位的状态，如果是开，则返回true。

bits.get(i);

1. set

设置某个位为开状态：

bits.set(i);

1. clear

将第i个位设为关状态：

bits.clear(i);