

# CS209A 计算机系统设计与应用

## 课程 2

### 2.1 泛型(从 JDK 5.0)

优点:

1. 无需类型转换 (类型安全)

名词:

```
1 List<E>; //泛型
2 E; //类型形式参数
3 List<String>; //参数化类型
4 String; //实际类型参数
5 List; //原生类型, 在JDK5.0前使用
6 // 我们应该避免使用原生类型
```

使用泛型

1. 泛型类

```
1 public class gen<T>{
2     ...
3     //在这里我们可以使用T类型或其他含T类型的数据结构
4 }
```

2. 泛型接口, 例如, Comparable<T>

```
1 public interface Comparable<T>
2 {
3     int compareTo<T> (T o);
4 }
```

那么我们应该重写使用它的方法

```
1 public myInt implements Comparable<myInt>, Comparable<myInt>{
2     //实现了对不同类型的比较
3     int data;
4     public int compareTo<myInt>(myInt m){
5         if(data == m.data){
6             return 0;
7         }else if(data > m.data){
8             return 1;
9         }else {
10            return -1;
11        }
12    }
```

```

12     }
13     public int compareTo<int>(int i){
14         if(data == i){
15             return 0;
16         }else if(data > i){
17             return 1;
18         }else {
19             return -1;
20         }
21     }
22 }

```

### 3. 泛型方法

```

1 public static <E> Set<E> union(Set<E> s1, Set<E> s2){
2     //类型参数应该在"static"之后声明
3     Set<E> result = new HashSet<>(s1); //<>中的类型可忽略
4     result.addAll(s2);
5     return result;
6 }

```

## 类型参数的边界

### 1. 格式

```

1 <T extends superclass & interface1 & interface2>;
2 //使用 & 分隔类和接口
3 //超类应放在第一个

```

### 2. 应用

```

1 public static<T extend Comparable> Pair<T> minmax(T[] a){
2     ...
3 }

```

## 通配符

### 1. 创建泛型类型之间的关系

```

1 String is a subclass of Object
2 List<String> is not a subclass of List<Object>

```

用于方法、类或接口中的参数匹配

```

1 public static void process(List<?> list){
2     //我们可以将 List<String> 传递给这个方法
3 }

```

### 2. 设置通配符的界限

```
1 //“继承”可以是直接的也可以是间接的
2 List<? extends superclass>;
3 //我们可以传递 List<T>, 其中 T 继承superclass
4 List<? super subclass>;
5 //我们可以传递 List<T>, 其中subclass继承了 T
```

## 类型擦除

T 转换为 Object。

JVM 中根本没有泛型。

## 2.2 抽象数据类型(ADT)

### 基本类型:

1. **值** 立即映射到机器码
2. **操作** 立即映射到机器码

### ADT:

1. 由一组值和一组操作定义的对象类型
2. **隐藏了**值在内存中的存储方式和操作的实现方式
3. 客户端只知道可以访问的数据选项。

### ADT的操作

1. creator: 创建该类型的新对象
2. producers: 从旧对象创建新对象
3. observer: 为当前ADT返回一个不同的类型
4. Mutators: 修改对象本身

### 2.3.1 集合

1. 一组对象
2. 主要用于数据存储、数据检索和数据操作

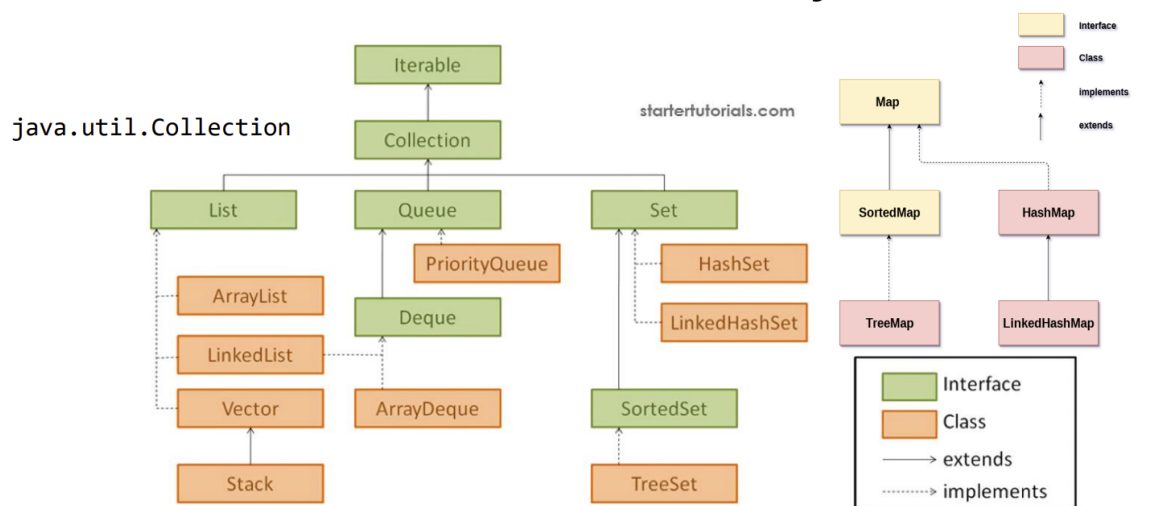
### 2.3.2 Java集合框架

#### 接口

## 1. 集合类层次结构

Java

# Collection Class Hierarchy



## 2. 可迭代<T> 接口:

```
1 //实现这个接口允许一个对象成为“foreach”语句的目标
2 public interface Iterable<T> //可迭代的，即一定会有一个迭代器
3 {
4     Iterator<T> iterator();
5 }
```

```
1 public interface Iterator<E>
2 {
3     boolean hasNext();
4     E next();
5     void remove(); //删除目前迭代器迭代到的元素
6     //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
7     //解决方法1: remove之后要调用iterator.next()
8     //解决方法2: 使用foreach,每迭代一次会自动调用迭代器方法
9 }
```

## 3. Collection 接口

```
1 public interface Collection<E>{
2     int size();
3     boolean isEmpty();
4     boolean contains(Object element);
5     //增加，删除元素不强制重写
6     boolean add(E element);//optional
7     boolean remove(Object element);//optional
8
9     //从一个假头开始
10    Iterator<E> iterator(); //返回该对象的迭代器
```

```

11
12     Object[] toArray(); //返回一个包含集合中所有元素的 Object 类型数组
13     /*
14     1.方法的参数 a 是用于指定数组类型和大小的数组。
15     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，
16     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。
17     */
18     E[] toArray(E a[]);
19
20     //Bulk Operations
21     boolean containsAll(Collection<?> c); //判断是否包含
22     boolean addAll(Collection<? extends E> c); //增加一系列的而元素
23     boolean removeAll(Collection<?> c); //删除当前集合与另一个集合c中相同的元素。
24
25     /*保留集合中与另一个集合 c 相同的元素，
26     而删除集合中不在另一个集合 c 中的元素。*/
27     boolean retainAll(Collection<?> c);
28
29     void clear(); //清空当前集合
30 }

```

#### 4. Set 接口

1. 没有在Collection基础上添加方法
2. 添加规定：无重复元素
3. 重新定义collections<E>或者objects的一些方法
4. Set 惯用语法

```

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

```

#### 5. List 接口

1. 列表有 "顺序"
2. 增加方法

```

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

```

#### 6. Map 接口

```

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

```

```

4    boolean containsKey(Object key); //判断是否包含一个键
5    boolean containsValue(Object value); //判断是否包含值
6    V get(Object key); //通过键取值
7    V put(K key, V value); //存放键值对, optional
8    V remove(Object key); //删除键值对并返回值, optional
9    void putAll(Map<? extends K, ? extends V> t); //optional
10   void clear();
11
12   //从一个collection的角度
13
14   public Set<K> keySet();
15   public Collection<V> values();
16
17   //返回所有键值对组成的集合
18   public Set<Map.Entry<K, V>> entrySet();
19 }

```

## 底层实现

### 1. List 底层实现

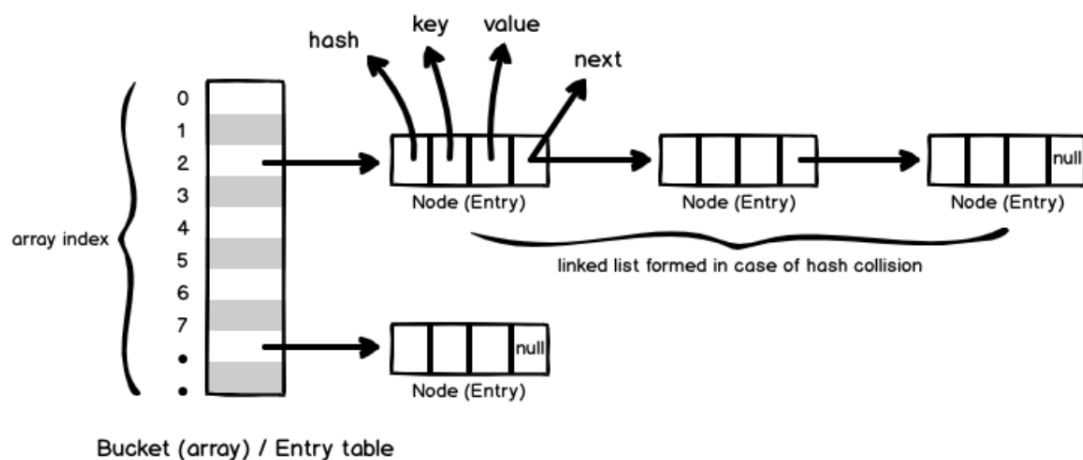
1. ArrayList 在内部使用数组来存储元素。

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 实现原理

#### 1. HashMap 结构



#### 2. Map 实现原理

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

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

### 3. hashCode() and equals()

1. `hashCode()` 将内部地址转换为 int 并返回
2. `equals(Objected)` 默认情况下比较哈希码。
3. `==` 总是比较哈希码