集合框架2

一 内容回顾（列举前一天重点难点内容）

1.1 教学重点:

```
1.掌握Lambda表达式的概念
2.掌握Lambda表达式的基本使用
3.熟悉Lambda表达式的使用场景
4.掌握集合的分类特点
5.掌握集合的常用方法
6.掌握集合的遍历
7.掌握迭代器的工作原理
8.掌握List的常用方法
9.掌握List的遍历排序
```

1.2 教学难点:

```
1.lambda的进阶理解
lambda表达式的熟练使用可以帮开发者节省大量的代码,提高编程效率,但是进阶语法的理解有一定的难度,比如函数引用.对于初学者应该先学会基本使用,在此基础上先记住进阶的语法,再慢慢理解使用方法.
2.迭代器的工作原理
对于迭代器最重要的是先会使用,能熟练实现集合遍历.工作原理方面可以先简单理解一下,能够帮助我们更好的掌握迭代器的使用,它使用的场景很多:比如for循环内部,数据库访问(jdbc)遍历元素等.而对原理的深入探究看个人精力,可以暂时放一下.
```

二 教学目标

```
1.熟练使用ArrayList和LinkedList
2.掌握数据结构分类
3.掌握常用数据结构的基本原理(数组,链表,栈,队列)
4.掌握泛型的作用
5.掌握泛型的基本使用(类上,方法上,接口上)
6.掌握工具类Collections的基本使用
```

三 教学导读

3.1. List详解

```
昨天我们学习了List的基本知识,今天我们继续学习List的主要子类使用
```

3.2. 泛型

3.2..1. 泛型的简介

泛型， 指的是“泛指的类型”。 将数据类型参数化。

使用泛型， 将某些类型， 在类与类、 类与接口、 方法之间进行传递。 类似于 “传参” 。

3.2..2. 泛型的好处

- 用在集合中， 限制存储的元素的类型， 不用再使用元素的时候， 逐个元素进行类型检查,简化代码.
- 可以提高代码的可读性。
- 可以使某些发生在运行时期的逻辑错误问题， 提前到编译时期,提高效率.

3.2..3. 泛型的定义方式

泛型， 是定义在一对尖括号里面的。 在尖括号里面定义一个类型。 此时，定义在这一对尖括号中的类型， 就是泛型。

- 泛型， 是一个标识符， 遵循大驼峰命名法。
- 泛型， 一般情况下， 不用太长的类型来描述。 一般情况下， 只需要使用一个字母代替即可。
- 如果需要定义多种泛型， 直接在尖括号中定义， 泛型与泛型之间以逗号分隔即可。

3.2..4. 泛型指定类型

在使用到泛型类、接口、方法的时候， 指派每一个泛型具体是什么类型。

注意事项： 泛型类型的指派， 只能是引用数据类型。 泛型不能设置为基本数据类型。 如果真的需要使用到基本数据类型， 使用他们对应的包装类。

四 教学内容

4.1. List详解

4.1.1. ArrayList与LinkedList对比(会)

- 相同点：
    - 都是List集合的常用的实现类。
    - 对集合中的元素操作的方法基本一致。

    - 都是线程不安全的
- 不同点：
    - ArrayList底层实现是数组， 使用数组这种数据结构进行数据的存储。
    - LinkedList底层实现是双链表， 使用双链表这种数据结构进行数据的存储。

数组与链表结果特点比较:

- 数组实现功能时查找快,添加删除慢
- 链表查找慢,添加删除快

ArrayList与LinkedList的使用场景：

- 如果对集合中的元素， 增删操作不怎么频繁， 查询操作比较频繁。 使用ArrayList。
- 如果对集合中的元素， 增删操作比较频繁， 查询操作不怎么频繁。 使用LinkedList。

4.1.2. 集合对自定义对象的存储(会)

- 类分成系统类和自定义类
- 系统类往往已经重写了tostring(),equals(),hashcode()等方法
- 自定义类,为了实现我们的功能,往往我们需要重写父类的常用方法,比如:tostring(),equals(),hashcode()

示例代码:

例题:将各学科名字(java,python,python,iOS,bigdata)存入集合,注意名字里面有重复,要求使用List存储数据,但是数据不能重复

分析:List默认是有序可重复的---利用Contains

```java
public class Demo6 {
    public static void main(String[] args) {
        test1();
    }

    public  static  void  test1(){
        ArrayList list1 = new ArrayList();
        list1.add("java");
        list1.add("python");
        list1.add("python");
        list1.add("iOS");
        list1.add("bigdata");
        System.out.println(list1);

        //创建临时集合
        ArrayList list2 = new ArrayList();

        Iterator i = list1.iterator();
        while (i.hasNext()){
            Object o = i.next();
            if (!(o instanceof  String)){
                throw  new ClassCastException();
            }

            //向下转型
            String s = (String)o;

            if (!list2.contains(s)){
                //当list1中不包换object时,将它添加进来
                /*
                 * 如果判断成立,说明list1中不包含当前的元素
                 * 工作原理:当添加元素时,会让当前的元素与集合中已有的元素通过equels方法进行一一比较.过程中
```

```
                    * 只要有一次返回true,停止比较.让整个的contains方法返回true.只有所有的比较都返回false,最终
                    * 才会返回false
                    *
                    * 实例:添加第三个元素的时候,调用equals方法的过程
                    * 第三元素.equals("java") = false     第三元素.equals("python") = true  停止比较
                    */
                    list2.add(s);
                }
            }
            System.out.println(list2);
        }


```

对于上题的要求,如果将课程列表换成人的对象,结果会怎么样呢?

示例代码:

将人类的对象存储到集合中,并按照年龄和姓名比较,相同则认为是同一个人

分析:

如果直接将Person对象存入list,不会按照年龄和姓名去重,因为Person没有重写Object类的equals方法,默认比较的是对象地址,所以此时需要重写equals方法,在方法内部按照年龄和姓名比较.

```
public  static  void  test2(){
        ArrayList list1 = new ArrayList();
        list1.add(new Person("zhangsan",20));
        list1.add(new Person("lisi",20));
        list1.add(new Person("lisi",20));
```

```
        list1.add(new Person("wangwu",204));

        System.out.println(list1);

        ArrayList list2 = new ArrayList();

        Iterator i = list1.iterator();
        while (i.hasNext()){
            Object o = i.next();
            if (!(o instanceof  Person)){
                throw  new ClassCastException();
            }

            //向下转型
            Person s = (Person)o;

            if (!list2.contains(s)){
                list2.add(s);
            }
        }
        System.out.println(list2);
    }

}

class  Person{
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
```

```
39
40      @Override
41      public String toString() {
42          return "Person{" +
43                  "name='" + name + '\'' +
44                  ", age=" + age +
45                  '}';
46      }
47
48      //重写equals方法
49
50      @Override
51      //自己制定的比较规则:并按照年龄和姓名比较,相同则认为是同一个
    人
52      public boolean equals(Object o) {
53          if (this == o) return true;
54          if (o == null || getClass() != o.getClass())
    return false;
55          Person person = (Person) o;
56          return age == person.age &&
57                  Objects.equals(name, person.name);
58      }
59
60  }
61
62
```

4.1.3. LinkedList(会)

对于LinkedList的特有方法对比:

jdk1.6以前的删除获取方法,拿不到元素报异常

jdk1.6以后的删除获取方法,拿不到元素返回null

所以建议使用1.6之后的.

```
public class Demo1 {
    public static void main(String[] args) {
        LinkedList linkedList = new LinkedList();

        //jdk1.6以前

     //addFirst()//始终在首位添加
     //addLast()//始终在末尾添加

        linkedList.add("java");
        linkedList.add(1,"php");
        linkedList.addFirst("python");
        linkedList.addLast("bigdata");
        linkedList.add(3,"html");
        linkedList.add(0,"c");
        linkedList.addFirst("c++");
        System.out.println(linkedList);// [c++, c, python, java, php, html, bigdata]

        //linkedList.clear();//清除所有数据

        //获取的对象不存在会发生异常 NoSuchElementException
        //E getFirst()
        //E getLast()
        //System.out.println(linkedList.getFirst());

        //删除的对象不存在会发生异常
        //E removeFirst()
        //E removeLast()

```

```
30          //从jdk1.6开始出现以下方法
31
32          //offerFirst()
33          //offerLast()
34
35          //获取的对象不存在会返回null
36          //peekFirst()
37          //peekLast()
38          System.out.println(linkedList.peekFirst());
39
40          //删除的对象不存在会返回null
41          //pollFirst()
42          //pollLast()
43
44      }
45  }
46
```

4.1.4. 数据结构(了解)

4.1.4.1. 数据结构定义

数据结构是计算机存储、组织数据的方式,是相互之间存在一种或多种特定关系的数据元素的集合，即带“结构”的数据元素的集合。“结构”就是指数据元素之间存在的关系，分为逻辑结构和存储结构.通常情况下，精心选择的数据结构可以带来更高的运行或者存储效率。

数据的逻辑结构和物理结构是数据结构的两个密切相关的方面，同一逻辑结构可以对应不同的存储结构。算法的设计取决于数据的逻辑结构，而算法的实现依赖于指定的存储结构.

4.1.4.2. 数据的逻辑结构

指反映数据元素之间的逻辑关系的数据结构，其中的逻辑关系是指数据元素之间的前后间关系，而与他们在计算机中的存储位置无关。逻辑结构包括：

- 集合：数据结构中的元素之间除了“同属一个集合” 的相互关系外，别无其他关系；
- 线性结构：数据结构中的元素存在一对一的相互关系；
- 树形结构：数据结构中的元素存在一对多的相互关系；
- 图形结构：数据结构中的元素存在多对多的相互关系。

4.1.4.3. 数据的物理结构

指数据的逻辑结构在计算机存储空间的存放形式。

4.1.4.4. 常用的数据结构

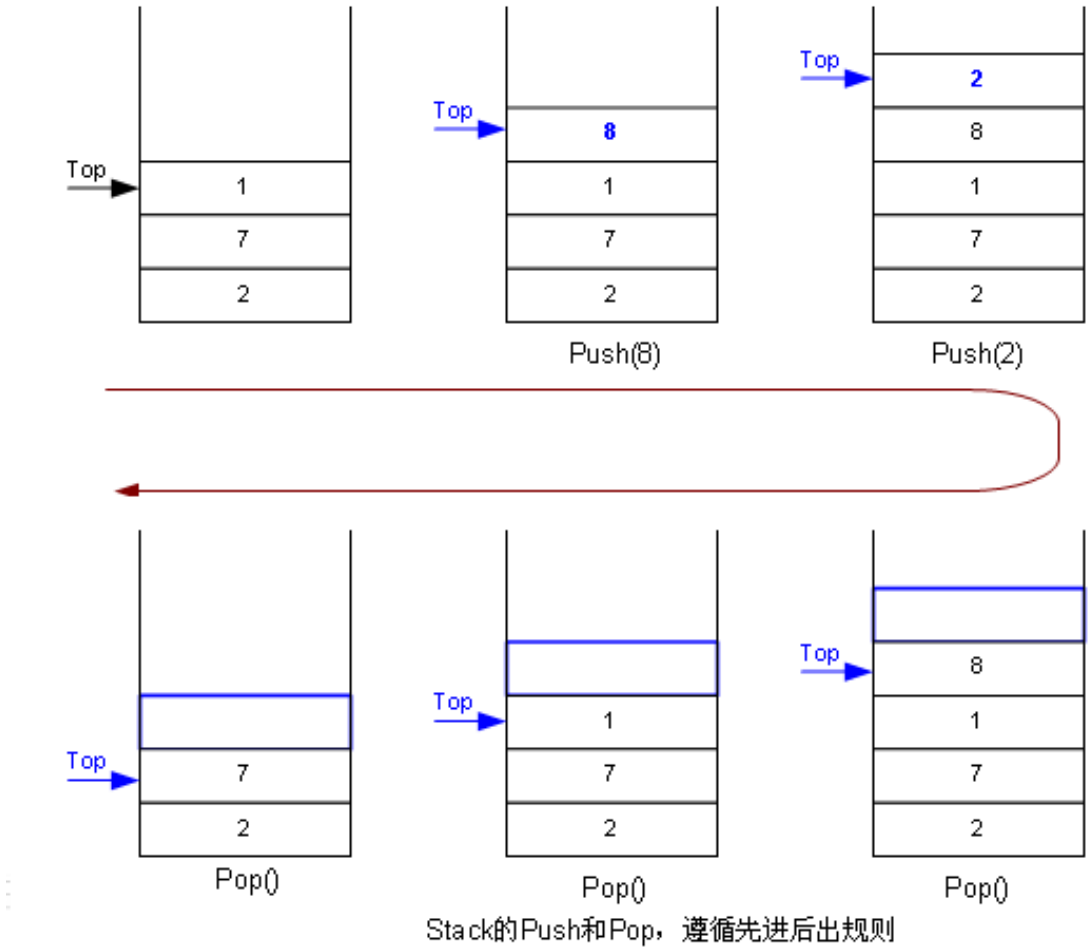
程序设计中常用的数据结构包括如下几个。注意:这里的数据结构分类是从逻辑结构上进行的.

- 数组(Array)

数组是一种聚合数据类型，它是将具有相同类型的若干变量有序地组织在一起的集合。数组可以说是最基本的数据结构，在各种编程语言中都有对应。一个数组可以分解为多个数组元素，按照数据元素的类型，数组可以分为整型数组、字符型数组、浮点型数组、指针数组和结构数组等。数组还可以有一维、二维以及多维等表现形式。

- 栈(Stack)

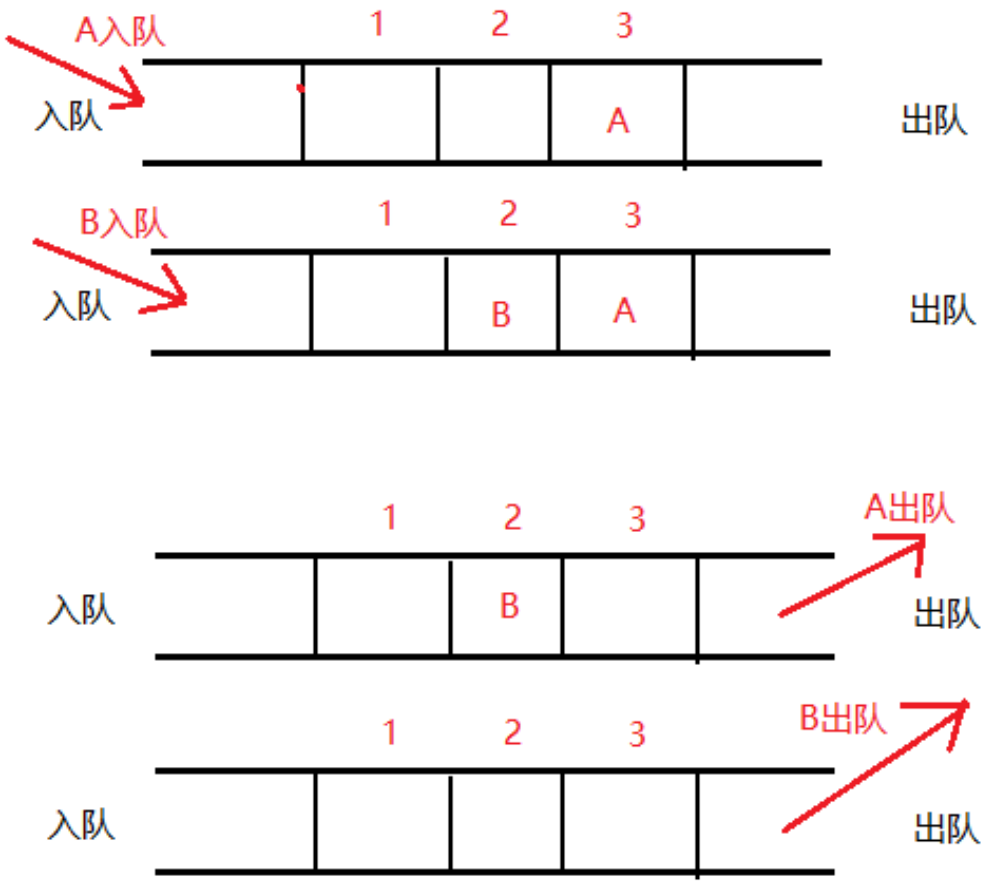
栈是一种特殊的线性表，它只能在一个表的一个固定端进行数据结点的插入和删除操作。栈按照后进先出的原则来存储数据，也就是说，先插入的数据将被压入栈底，最后插入的数据在栈顶，读出数据时，从栈顶开始逐个读出。栈在汇编语言程序中，经常用于重要数据的现场保护。栈中没有数据时，称为空栈。



Stack的Push和Pop，遵循先进后出规则

- 队列(Queue)

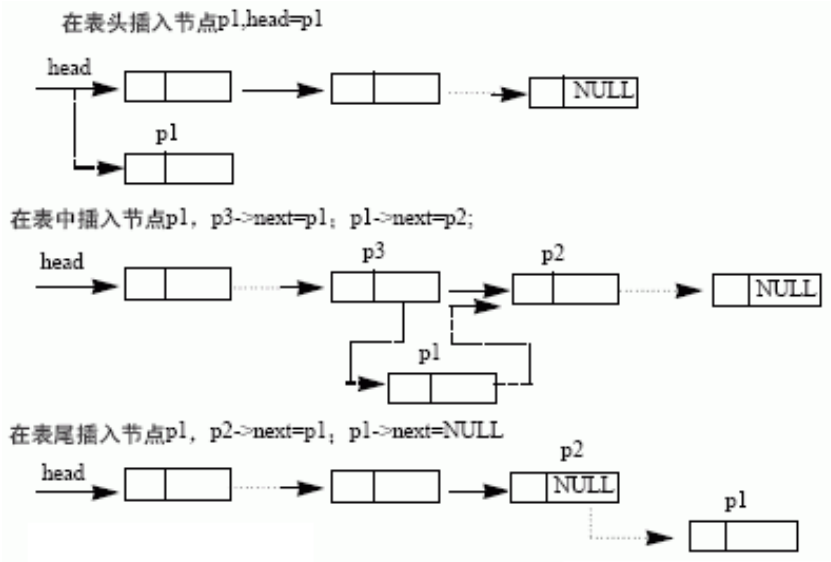
队列和栈类似，也是一种特殊的线性表。和栈不同的是，队列只允许在表的一端进行插入操作，而在另一端进行删除操作。一般来说，进行插入操作的一端称为队尾，进行删除操作的一端称为队头。队列中没有元素时，称为空队列。



Queue,遵循先进先出原则

- 链表(Linked List)

链表是一种数据元素按照链式存储结构进行存储的数据结构，这种存储结构具有在物理上存在非连续的特点。链表由一系列数据结点构成，每个数据结点包括数据域和指针域两部分。其中，指针域保存了数据结构中下一个元素存放的地址。链表结构中数据元素的逻辑顺序是通过链表中的指针链接次序来实现的。



单链表中插入节点

- 树(Tree)

树是典型的非线性结构，它是包括，2个结点的有穷集合K。在树结构中，有且仅有一个根结点，该结点没有前驱结点。在树结构中的其他结点都有且仅有一个前驱结点，而且可以有两个后继结点，m≥0。

- 图(Graph)

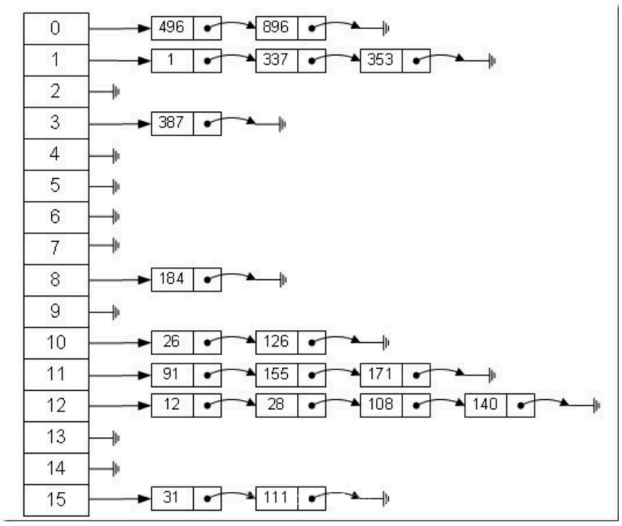
图是另一种非线性数据结构。在图结构中，数据结点一般称为顶点，而边是顶点的有序偶对。如果两个顶点之间存在一条边，那么就表示这两个顶点具有相邻关系。

- 堆(Heap)

堆是一种特殊的树形数据结构，一般讨论的堆都是二叉堆。堆的特点是根结点的值是所有结点中最小的或者最大的，并且根结点的两个子树也是一个堆结构。

- 散列表(Hash)

散列表源自于散列函数(Hash function)，其思想是如果在结构中存在关键字和T相等的记录，那么必定在F(T)的存储位置可以找到该记录，这样就可以不用进行比较操作而直接取得所查记录。



4.2. 泛型(会)

4.2.1. 泛型在类中的使用(会)

4.2.1.1. 语法部分

定义： 在类名的后面， 紧跟上一对尖括号。

```
class Animal <T> {}
class Dog <T, M> {}
```

泛型类的使用： 声明引用、实例化对象、被继承。

```
// 1. 声明引用
Animal<String> animal;
// 2. 实例化对象
Animal<Integer> animal = new Animal<>();
// 3. 被继承
class Dog extends Animal<String> {}
class Dog<T> extends Animal<T> {}
```

4.2.1.2. 泛型类的特点

- 在类中定义的泛型， 虽然还不明确是什么类型， 但是在当前类中是可以使用的。
- 在使用到这个类的时候， 必须要指定泛型的类型。 如果不指定， 默认是 Object。
- 泛型， 只能在当前的类中使用， 不能在其他的类中使用， 包括子类。

4.2.1.3. 示例代码

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Test {
    public static void main(String[] args) {
        // 实例化一个对象
        // 指定了泛型的类型是 String 类型
        Person<String> xiaoming = new Person<>();
        // 泛型，是在编译前期进行的类型检查。一旦编译完成，泛型就不存在了。
        xiaoming.part = "xiaoming";

        Person<Integer> xiaohong = new Person<>();
        xiaohong.part = 2;

        // 实例化一个子类对象
        Student student = new Student();
        student.part = "abc";

        // 实例化一个泛型子类对象
```

```
        Teacher<Integer> xiaowang = new Teacher<>();
        xiaowang.part = 132;

        // 如果多个泛型，使用这个类的时候，逐个进行类型指派即可
        Animal<String, Integer> animal = new Animal<>();

        // 如果对于一个泛型类，在使用的时候，没有指派类型，默认是 Object 类型
        Person xiaobai = new Person();
    }
}

class Animal<T, M> { }

class Person<T> {
    String name;
    int age;
    //在类上确定的泛型可以直接在类的内部使用
    T part;      // 虽然现在还不明确T是什么类型，但是我们可以使用这个类型。
}

class Student extends Person<T> {}

class Teacher<T> extends Person<T> { }
```

课上练习

学生使用工具(Tools类),用手机电话和用电脑玩游戏.

```
public class Demo7 {
```

```
    public static void main(String[] args) {
        Student student = new Student();
        Phone phone = new Phone("华为");

        Computer computer = new Computer("小米");
        student.setTools(computer);//  tools = phone 多态
        Tools tools = student.getTools();  // 多态
        //使用泛型前
        if (tools instanceof  Phone){
            Phone phone1 = (Phone)tools;
            phone1.callPhone();
        }else {
            throw new ClassCastException("不是Phone类");
        }

        //使用泛型后
        Student1<Computer> student1 = new Student1();
//          student1.setTools(phone);
    }
}

//使用泛型前
class Student{
    Tools tools;

    public Tools getTools() {
        return tools;
    }

    public void setTools(Tools tools) {
        this.tools = tools;
    }
```

```
}

//使用泛型后
/*
 * 给Student1类加泛型,方式:在类的后面直接添加<E>,E代表任意一种数据类型,注意:这里不一定是E,任意字母都可以
 * 这就是在类上使用泛型
 * 在类上确定的泛型可以直接在方法上使用
 *
 * 当泛型没有指定时默认是Object
 */
class Student1<E>{
    E tools;
    public E getTools() {
        return tools;
    }

    public void setTools(E tools) {
        this.tools = tools;
    }
}

class  Tools{
    String name;
}

class Phone extends  Tools{
    public Phone(String name) {
        this.name = name;
    }

    public  void  callPhone(){
        System.out.println("打电话");
```

```
    }
}

class  Computer extends  Tools{
    public Computer(String name) {
        this.name = name;
    }

    public  void  play(){
        System.out.println("play");
    }
}
```

4.2.2. 泛型在接口中的使用(会)

4.2.2.1. 语法部分

泛型接口的定义： 在接口名字的后面， 添加上一对尖括号。 在尖括号里面定义泛型。

```
interface MyInterface<T> {}
interface MyInterface<T, M> {}
```

泛型接口的使用： 实现类实现接口、 使用接口访问接口中的静态成员、 被继承。

```
// 1. 实现类实现接口
class MyInterface1Impl implements MyInterface1<Person>
{}
// 2. 被继承
interface SubMyInterface extends MyInterface1<String> {}
```

4.2.2.2. 泛型接口的特点

- 在接口中定义的泛型， 虽然还不明确是什么类型， 但是在当前接口中是可以使用的。
- 在使用到这个接口的时候， 必须要指定泛型的类型。 如果不指定， 默认是 Object。
- 泛型， 只能在当前的接口中使用， 不能在其他的接口中使用， 包括子接口。

示例代码

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description 泛型接口的使用
 */
public class Test {
    public static void main(String[] args) {
        // 使用匿名内部类的形式实现接口
        MyInterface1<Person> impl = new
MyInterface1<Person>() {
            @Override
            public int compareTo(Person o1, Person o2)
{
```

```
                return 0;
            }
        };
        // 使用lambda表达式实现接口
        // 此时，会根据左侧的接口引用中的泛型，推导出接口实际指派的类型是谁
        MyInterface1<Person> impl1 = (o1, o2) -> o1.age - o2.age;
        MyInterface1<String> impl2 = (o1, o2) -> o1.length() - o2.length();
    }
}

// 定义一个泛型接口
interface MyInterface1<T> {
    int compareTo(T o1, T o2);
}

// 子接口
interface SubMyInterface extends MyInterface1<String> {}

// 实现类
class MyInterface1Impl implements MyInterface1<Person> {
    @Override
    public int compareTo(Person o1, Person o2) {
        return 0;
    }
}

class Person {
    int age;
```

```
}
```

4.2.2.3 子类使用接口泛型

这里有两种情况:

第一种:子类与接口泛型一致,接口可以直接使用泛型类型

第二种:接口使用泛型,子类不用,在实现的接口位置必须指定一个具体的数据类型

```java
public class Demo9 {
    public static void main(String[] args) {
        Bird<String> bird = new Bird();
        bird.show("haha");
    }
}

interface  Inter<E>{
    public void  show(E e);
}
//1.子类与接口一致
/* 类上的泛型确定了,接口上的泛型就确定了,方法上的泛型就确定了
 */
class  Bird<E> implements  Inter<E>{
    @Override
    public void show(E e) {

    }
}

//2.接口使用泛型,子类不用
/*在实现的接口位置必须指定一个具体的泛型
 *
```

```
 * 方法使用泛型的情况:
 * 1.如果是重写的方法,泛型与接口一致
 * 2.如果是子类自己的方法,可以与接口一致,也可以有自己的泛型
 */
class  Cat implements  Inter<String>{
    @Override
    public void show(String s) {

    }
}

class  Pig implements  Comparable<Pig>{
    @Override
    public int compareTo(Pig o) {
        return 0;
    }
}

class ComWithA implements Comparator<String>{
    @Override
    public int compare(String o1, String o2) {
        ArrayList list = null;
        return 0;
    }
}
```

4.2.3. 泛型在方法中的使用(会)

4.2.3.1. 语法部分

定义： 泛型方法中， 在定义方法时,在返回值前面通过定义泛型。

```
public static <T> void test(T t) {

}
```

- 在定义处的<>用来定义泛型
- 在调用处的<>用来使用泛型

4.2.3.2. 泛型方法的分类

- 第一:方法上的泛型与类上的一致
- 第二:方法上独立使用泛型
    - 在方法中定义的泛型， 虽然还不明确是什么类型， 但是在当前方法中是可以使用的。
    - 泛型方法， 在使用的时候， 不能跟类、接口似的， 手动的设置类型。 泛型方法中， 泛型的设置， 在参数中体现。
    - 泛型方法， 一定需要是有参的。 参数列表中， 必须有泛型类型。
    - 泛型方法中的泛型的设置， 是通过在调用方法的时候， 实参的类型推导出来的。
    - 泛型， 只能在当前的方法中使用， 不能在其他的方法中使用。
- 第三:静态方法使用泛型

4.2.3.3. 示例代码

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
```

```java
public class Demo8 {
    public static void main(String[] args) {
        Dog<String> dog = new Dog();
        //第一:方法上的泛型与类上的一致
        dog.run("跑");
        //第二:方法上独立使用泛型
        dog.show(3);
        //第三:静态方法使用泛型
        Dog.eat("java");
    }
}
class Animal<E>{
}

class  Dog{
    //第一:方法上的泛型与类上的一致
    public   void run(E e){
        System.out.println(e);
    }
    //第二:方法上独立使用泛型
    /*
     * 注意:泛型在使用之前一定要先进行定义
     * 定义的方式:在当前方法的最前面添加<泛型>
     * 作用:让方法与方法内的泛型保持一致
     */
    public <F>  void  show(F f){
        ArrayList<F> list = new ArrayList();
    }
    //第三:静态方法使用泛型
    /*
     * 必须独立使用
     * 方式:在static 后面定义泛型  <泛型>
     *
```

```
     * 不能使用类上定义的泛型
     */
    public static <W>  void  eat(W e){
    System.out.println(e);
    }

}
```

4.2.4. 关于java中?的使用总结(了解)

- 用于?: 这里是三目运算符一部分,前面是判断条件,后面是两个分支结果
- 用于数据库的sql语句 select * from emp where name=? :表示占位符
- 用于泛型,是通配符,表示任意一种数据类型.

```
  //这里的Object与前面的?没有关系
  Class<?> class1 = Object.class;

  //如果类的泛型使用时写成?是可以的.作为返回值时,会默认成Object类型,但是作为参数不行.
    //所以在给对象指定泛型时要写具体类型
  Test<?> test = new Test();

  //test.run(new Object());
class Test<T>{
   T e;
  public T run(T a) {
    T t = null;
    return t;
  }
}
```

4.2.5. 限制上限<? extends E>(了解)

定义:限制的是整个的<>可以取的泛型类型的上限是E,<>中可以取的类型是E及E的子类

示例代码

```
public class Demo10
{
    public static void main(String[] args) {
        //
        ArrayList<Student2> list1 = new ArrayList<>();
        list1.add(new Student2("bingbing", 1));
        //可以传参:因为Student2是Person4的子类,可以实现遍历
        bianli(list1);

        ArrayList<Teacher> list2 = new ArrayList<>();
        list2.add(new Teacher("bingbing", 1));
        //可以传参:因为Teacher是Person4的子类,可以实现遍历
        bianli(list2);

        ArrayList<Person4> list3 = new ArrayList<>();
        list3.add(new Person4("bingbing", 1));
        //可以传参
        bianli(list3);

        ArrayList<Object> list4 = new ArrayList<>();
        list4.add(new Object());
        //可以传参:因为Object是Person4的父类,不可以实现遍历
        //bianli(list4);
    }


```

```
27      public static void bianli(Collection<? extends
    Person4> e){
28          System.out.println("遍历了");
29      }
30  }
31
32  class Person4{
33      String name;
34      int age;
35      public Person4() {
36          super();
37          // TODO Auto-generated constructor stub
38      }
39      public Person4(String name, int age) {
40          super();
41          this.name = name;
42          this.age = age;
43      }
44      @Override
45      public String toString() {
46          return "Person4 [name=" + name + ", age=" + age
    + "]";
47      }
48  }
49
50  class Teacher extends Person4{
51      public  Teacher(String name, int age) {
52          super(name, age);
53      }
54  }
55
56  class Student2 extends Person4 {
57      public  Student2(String name, int age) {
```

```
58          super(name, age);
59      }
60  }
```

4.2.6. 限制下限<? super E>(了解)

定义:限制的是整个的<>可以取的泛型类型的下限是E,<>中可以取的类型是E及E的父类

示例代码

```
1  public class Demo11 {
2      public static void main(String[] args) {
3          //限制下限 <? super E>
4          //TreeSet(Comparator<? super E> comparator) :这里的E跟TreeSet后面的泛型类型一致,所以现在E应该表示的Student3
5
6          //创建Student3类的比较器对象
7          ComWithStu comWithStu = new ComWithStu();
8          //创建Teacher1类的比较器对象
9          ComWithTea comWithTea = new ComWithTea();
10         //创建Person1类的比较器对象
11         ComWithPerson1 comWithPerson = new ComWithPerson1();
12         //创建GoodStudent类的比较器对象
13         ComWithGood comWithGood = new ComWithGood();
14
15         TreeSet<Student3> set = new TreeSet<>(comWithStu);//因为这里限制的是Student3及他的父类
16         //TreeSet<Student3> set = new TreeSet<>(comWithTea);//不可以使用,因为Teacher2类与Student3类没有关系
17         //TreeSet<Student3> set = new TreeSet<>(comWithPerson);//可以  ,因为Person3类是Student3类的父类
```

```
18          //TreeSet<Student3> set = new TreeSet<>
(comWithGood);//不可以,因为GoodStudent类是Student3类的子类
19          set.add(new Student3("bingbing"));
20          set.add(new Student3("bingbing1"));
21          set.add(new Student3("bingbing2"));
22      }
23  }
24
25  //创建Student3类的比较器
26  class ComWithStu implements Comparator<Student3> {
27
28      public int compare(Student3 o1, Student3 o2) {
29
30          return o1.name.compareTo(o2.name);
31      }
32  }
33  //创建Teacher2类的比较器
34  class ComWithTea implements Comparator<Teacher2>{
35      public int compare(Teacher2 o1, Teacher2 o2) {
36
37          return 0;
38      }
39  }
40  //创建Person2类的比较器
41  class ComWithPerson1 implements Comparator<Person3>{
42      @Override
43      public int compare(Person3 o1, Person3 o2) {
44          // TODO Auto-generated method stub
45          return 0;
46      }
47  }
48
49  //创建GoodStudent类的比较器
```

```
class ComWithGood implements Comparator<GoodStudent>{
    public int compare(GoodStudent o1, GoodStudent o2)
{
        return 0;
    }
}


class Person3{
    String name;

    public Person3(String name) {
        super();
        this.name = name;
    }

    public String toString() {
        return "Person3 [name=" + name + "]";
    }
}

class Teacher2 extends Person3{
    public Teacher2(String name) {
        super(name);
    }
}
class Student3 extends Person3{
    public Student3(String name) {
        super(name);
    }
}

class GoodStudent extends Student3{
```

```
82      public GoodStudent(String name) {
83          super(name);
84      }
85  }
```

4.3. Collections工具类(会)

4.3.1. API

返回值	方法	描述
static <T> boolean	addAll(Collection<? super T> c, T... elements)	批量的向一个集合中添加数据。
static <T extends Object & Comparable<? super T>> T	max(Collection<? extends T> coll)	获取集合中最大的元素。
static <T> T	max(Collection<? extends T> coll, Comparator<? super T> comp)	获取集合中最大的元素。
static <T extends Object & Comparable<? super T>> T	min(Collection<? extends T> coll)	获取集合中最小的元素。
static <T> T	min(Collection<? extends T> coll, Comparator<? super T> comp)	获取集合中最小的元素。
static void	shuffle(List<?> list)	将集合中的元素随机排列。
static void	swap(List<?> list, int i, int j)	交换集合中两个元素。
static void	reverse(List<?> list)	将集合中的元素倒序。

static <T extends Comparable<? super T>> void	sort(List<T> list)	将集合中的元素升序排序。
static void	sort(List<T> list, Comparator<? super T> c)	将集合中的元素升序排序。
static int	binarySearch(List<? extends Comparable<? super T>> list, T key)	使用二分查找法查询元素下标。
static int	binarySearch(List<? extends T> list, T key, Comparator<? super T> c)	使用二分查找法查询元素下标。
static void	copy(List<? super T> dest, List<? extends T> src)	将src集合中的元素拷贝到dest中。
static void	fill(List<? super T> list, T obj)	使用指定的值填充集合。
static Collection	synchronizedCollection(Collection c)	获取一个线程安全的集合。
static Set	synchronizedSet(Set s)	获取一个线程安全的集合。
static List	synchronizedList(List list)	获取一个线程安全的集合。
static <K,V> Map<K,V>	synchronizedMap(Map<K,V> m)	获取一个线程安全的集合。

4.3.2. 示例代码

```
package day22.dCollections;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.List;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class CollectionsUsage {
    public static void main(String[] args) {
        // 1. 实例化一个List集合对象
        List<Integer> list = new ArrayList<>();

        // 2. 添加元素
        Collections.addAll(list, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0);

        // 3. 获取一个集合中的最大值，大小比较通过元素对应的类实现的Comparable接口进行比较
        Integer max = Collections.max(list);
        //    获取一个集合中的最大值，大小比较通过第二个参数Comparator
        Integer max2 = Collections.max(list, (i1, i2) -> i2 - i1);

        // 4. 获取一个集合中的最小值，大小比较通过元素对应的类实现的Comparable接口进行比较
```

```java
        Integer min = Collections.min(list);
        //    获取一个集合中的最小值，大小比较通过第二个参数Comparator
        Integer min2 = Collections.min(list, (i1, i2) -> i2 - i1);

        // 5. 将List集合中的数据进行随机的排列（洗牌）
        Collections.shuffle(list);

        // 6. 交换一个List集合中两个下标对应的元素
        Collections.swap(list, 0, 2);

        // 7. 将一个List集合中的元素倒序排列
        Collections.reverse(list);

        // 8. 将一个List集合进行排序，元素的大小比较规则使用元素对应的类实现的Comparable接口进行比较大小
        Collections.sort(list);
        //    将一个List集合按照指定的规则进行升序排序，基本不用，List集合中本身就有这样的排序方法
        Collections.sort(list, (i1, i2) -> i2 - i1);

        // 9. 集合中的元素拷贝，将作为第二个参数的集合中的数据拷贝到第一个集合中
        List<Integer> copy = new ArrayList<>();
        Collections.addAll(copy, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
        Collections.copy(copy, list);

        // 10. 使用指定的数据，填充一个集合
        Collections.fill(list, 0);

        // 11. 将线程不安全的集合，转成线程安全的集合
```

```
54          // Collections.synchronizedCollection()
55          // Collections.synchronizedList()
56          // Collections.synchronizedSet()
57          // Collections.synchronizedMap()
58      }
59  }
60
```