### ●集合框架(去除ArrayList中重复字符串元素方式)(掌握)

A:案例演示

需求：ArrayList去除集合中字符串的重复值(字符串的内容相同)

思路：创建新集合方式

public class Demo1\_ArrayList {  
 public static void main(String[] args) {  
 ArrayList list=new ArrayList();  
 list.add("a");  
 list.add("a");  
 list.add("b");  
 list.add("b");  
 list.add("c");  
 list.add("c");  
  
 ArrayList newList=*getSingle*(list);  
 System.*out*.println(newList);  
  
 }  
  
  
 *//案例 需求：ArrayList去除集合中字符串的重复值(字符串的内容相同)  
 //创建新集合将重复元素去掉  
 //明确返回类型ArrayList  
 //明确参数列表ArrayList  
 //1、创建新集合  
 //2、根据传入的集合(老集合)获取迭代器  
 //3、遍历老集合  
 //4、通过新集合判断是否包含老集合中的元素 不包含就添加* public static ArrayList getSingle(ArrayList list){  
 ArrayList newlist=new ArrayList();  
 *//老集合获取迭代器* Iterator it=list.iterator();  
 *//遍历老集合* while (it.hasNext()){  
 *//老集合遍历出来了之后记录* Object obj=it.next();*//记录每个元素  
 //如果新集合里没有老集合里元素* if (!newlist.contains(obj)){  
 *//就添加* newlist.add(obj);  
 }  
 }  
 *//返回新集合* return newlist;  
 }

|  |
| --- |
| **package** com.sxt.fuxi;  **import** java.util.ArrayList; **import** java.util.Iterator;  **public class** Demo9 {  **public static void** main(String[] args) {  ArrayList list = **new** ArrayList();  list.add(4); *//add方法往集合中添加数据通通都会自动提升为Object类型* list.add(5);  list.add(5);  list.add(6);  list.add(6);  list.add(4);  *//取出集合中重复的数据* ArrayList list2= **new** ArrayList();  Iterator it = list.iterator();  **while**(it.hasNext()){  Object obj = it.next();  **if**(!list2.contains(obj)){ *//判断新的集合中是否包含元素,不包含就add进去* list2.add(obj);  }  }  System.***out***.println(list2);   } } |

### ●集合框架(去除ArrayList中重复自定义对象元素)(掌握)

A:案例演示

需求：ArrayList去除集合中自定义对象元素的重复值(对象的成员变量值相同)

B:注意事项

remove()与contains（）方法的源码底层是用equals来比较

重写Student里面的equals()方法来解决问题

|  |
| --- |
| **package** com.sxt.fuxi;  **import** java.util.ArrayList; **import** java.util.Iterator;  **public class** Demo9 {  **public static void** main(String[] args) {  ArrayList list = **new** ArrayList();  list.add(**new** Student(**"小明"**,19)); *//add方法往集合中添加数据通通都会自动提升为Object类型* list.add(**new** Student(**"小明"**,19)); *//add方法往集合中添加数据通通都会自动提升为Object类型* list.add(**new** Student(**"小红"**,20)); *//add方法往集合中添加数据通通都会自动提升为Object类型* list.add(**new** Student(**"小红"**,20)); *//add方法往集合中添加数据通通都会自动提升为Object类型* list.add(**new** Student(**"高晓松"**,30)); *//add方法往集合中添加数据通通都会自动提升为Object类型* list.add(**new** Student(**"高晓松"**,30)); *//add方法往集合中添加数据通通都会自动提升为Object类型  //取出集合中重复的数据* ArrayList list2= **new** ArrayList();  Iterator it = list.iterator();  **while**(it.hasNext()){  Object obj = it.next();  **if**(!list2.contains(obj)){ *//判断新的集合中是否包含元素,不包含就add进去* list2.add(obj);  }  }  System.***out***.println(list2);   } } |
| **package** com.sxt.fuxi; **class** Student{  **private** String **name**;  **private int age**;   **public** Student() {  }   **public** Student(String name, **int** age) {  **this**.**name** = name;  **this**.**age** = age;  }   **public** String getName() {  **return name**;  }   **public int** getAge() {  **return age**;  }   @Override  **public** String toString() {  **return "Student{"** +  **"name='"** + **name** + **'\''** +  **", age="** + **age** +  **'}'**;  }    } |
| 以上代码发现并不能去重,查看contains的源码发现,底层用的equals对比对象是否重复  *父类Object默认是对比对象的内存地址 所以即使名字和年龄一样 地址不一样也会被集合添加进去* |
| **解决:重写Student类equals方法**  **package** com.sxt.fuxi; **class** Student{  **private** String **name**;  **private int age**;   **public** Student() {  }   **public** Student(String name, **int** age) {  **this**.**name** = name;  **this**.**age** = age;  }   **public** String getName() {  **return name**;  }   **public int** getAge() {  **return age**;  }   @Override  **public** String toString() {  **return "Student{"** +  **"name='"** + **name** + **'\''** +  **", age="** + **age** +  **'}'**;  }    *//父类Object默认是对比对象的内存地址,重写父类的equals方法根据我们的要求来对比对象中的属性作为衡量的重复要求* **public boolean** equals(Object obj) {  Student stu = (Student)obj; **boolean** boo1 = stu.getName().equals(**this**.getName());*//String 类中的equals* **boolean** boo2 = stu.getAge()== **this**.getAge();  **return** boo1 && boo2;  } } |
|  |

**模仿上面的案列完成：**

ArrayList集合中添加几个重复的Woker对象，

woker对象有属性姓名，年龄，Id

完成去重

|  |
| --- |
| **package** com.sxt.list;  **import** java.util.ArrayList; **import** java.util.Iterator; **import** java.util.Objects;  **public class** Demo1 {  **public static void** main(String[] args) {  ArrayList list= **new** ArrayList();  list.add(**new** Woker(**"张工"**,20,101));  list.add(**new** Woker(**"张工"**,20,101));  list.add(**new** Woker(**"李工"**,22,102));  list.add(**new** Woker(**"李工"**,22,102));  ArrayList list2 = **new** ArrayList();  Iterator it = list.iterator();  **while**(it.hasNext()){  Object obj =it.next();  **if**(!list2.contains(obj)){  list2.add(obj);  }  }  System.***out***.println(list2);  } } **class** Woker{  **private** String **name**;  **private int age**;  **private int id**;   **public** Woker() {  }   **public** Woker(String name, **int** age, **int** id) {  **this**.**name** = name;  **this**.**age** = age;  **this**.**id** = id;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public int** getAge() {  **return age**;  }   **public void** setAge(**int** age) {  **this**.**age** = age;  }   **public int** getId() {  **return id**;  }   **public void** setId(**int** id) {  **this**.**id** = id;  }   @Override  **public** String toString() {  **return "Woker{"** +  **"name='"** + **name** + **'\''** +  **", age="** + **age** +  **", id="** + **id** +  **'}'**;  }   */\* @Override  public boolean equals(Object obj) {  Woker w = (Woker)obj;  boolean boo1 = w.getName().equals(this.getName());  boolean boo2 = w.getAge() == this.getAge();  boolean boo3 = w.getId() == this.getId();  return boo1 && boo2 && boo3;  }\*/   //这是idea帮我们自动生成的重写equals方法* **public boolean** equals(Object o) {  System.***out***.println(**"equals被调用了"**);  **if** (**this** == o) **return true**; *//对比的内存地址* **if** (o == **null** || getClass() != o.getClass()) **return false**;  Woker woker = (Woker) o;  **return age** == woker.**age** &&  **id** == woker.**id** &&  Objects.*equals*(**name**, woker.**name**); *//等同于this.getName.equals(woker.getName)* }   } |
| 开发中推荐用idea自动生成equals方法,比我们自己写的要健壮一些 |

### ●集合框架(LinkedList的特有功能)(掌握)

A:LinkedList类概述

B:LinkedList类特有功能

public void addFirst(E e)及addLast(E e)

public E getFirst()及getLast()

public E removeFirst()及public E removeLast()

public E get(int index);

LinkedList 的数据结构是链子，所以get(index i)方法要判断从前面还是后面寻找

#### 了解:get(index i)方法要判断从前面还是后面寻找

|  |
| --- |
| Node<E> node(**int** index) {  *// assert isElementIndex(index);* **if** (index < (**size** >> 1)) {  Node<E> x = **first**;  **for** (**int** i = 0; i < index; i++)  x = x.**next**;  **return** x;  } **else** {  Node<E> x = **last**;  **for** (**int** i = **size** - 1; i > index; i--)  x = x.**prev**;  **return** x;  } } |
| get(index i)方法底层会判断从链表的左边还是右边查找  i< size/2来决定 |
| **package** com.sxt.list;  **import** java.util.Iterator; **import** java.util.LinkedList;  **public class** Demo2 {  **public static void** main(String[] args) {  LinkedList list= **new** LinkedList();  list.add(**"你好"**);  list.add(**'a'**);  list.addFirst(3);  list.addLast(56.8);  System.***out***.println(list);  Object obj =list.getFirst(); *//获取第一个元素* Object obj2 = list.getLast(); *//获取第最后的元素* System.***out***.println(obj);  System.***out***.println(obj2);  System.***out***.println(**"-------"**);   *//遍历* Iterator it = list.iterator();  **while**(it.hasNext()){  Object ob = it.next();  System.***out***.println(ob);  }  System.***out***.println(**"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**);  *//遍历2* **for** (**int** i = 0; i <list.size() ; i++) {  Object ob = list.get(i);*//* System.***out***.println(ob);  }  } } |
| 用的频率不高 |

### ●集合框架(List的三个子类的特点)

A:List的三个子类的特点

|  |
| --- |
| ArrayList:  底层数据结构是数组，查询、修改快，增删慢。  线程不安全，效率高。  Vector:  底层数据结构是数组，查询、修改快，增删慢。  线程安全，效率低。  Vector相对ArrayList查询慢(线程安全的)  Vector相对LinkedList增删慢(数组结构)  LinkedList:  底层数据结构是链表，查询、修改慢，增删快。  线程不安全，效率高。  Vector和ArrayList的区别  Vector是线程安全的,效率低  ArrayList是线程不安全的,效率高  共同点:都是数组实现的  ArrayList和LinkedList的区别  ArrayList底层是数组结构,查询和修改快  LinkedList底层是链表结构的,增和删比较快,查询和修改比较慢  共同点:都是线程不安全的 |

B:List有三个儿子，我们到底使用谁呢?

如果不知道用哪个,就用ArrayList

### ●集合框架(栈和队列数据结构)(掌握)

这些操作允许将链接列表用作堆栈(子弹夹)、[队列](mk:@MSITStore:F:\\MacBook\\课\\01-JAVASE\\00-API文档\\JDK%201.6%20API.chm::/java/util/../../java/util/Queue.html" \o "java.util 中的接口)(管道)

此类实现 Deque 接口，为 add、poll 提供先进先出队列操作

栈

先进后出

队列

先进先出

### ●集合框架(用LinkedList模拟栈数据结构的集合并测试)(掌握)

A:案例演示

需求：请用LinkedList模拟栈数据结构的集合，并测试

添加数据，先进的后出，后进的先出

LinkedList list=new LinkedList();  
 list.addLast("a");  
 list.addLast("b");  
 list.addLast("c");  
 list.addLast("d");  
  
/\* System.out.println(list.removeLast());  
 System.out.println(list.removeLast());  
 System.out.println(list.removeLast());  
 System.out.println(list.removeLast());\*/  
  
 //当栈中不为空就一直取数据  
 while (!list.isEmpty()){  
 System.*out*.println(list.removeLast());  
 }

题目要求：模拟栈数据结构的集合 应该创建栈对象 而不是LinkedList对象

|  |
| --- |
| **package** com.sxt.list;  **import** java.util.Iterator; **import** java.util.LinkedList;  **public class** Demo2 {  **public static void** main(String[] args) {  Stack s =**new** Stack();  s.in(**"aabc"**);  s.in(**'g'**);  s.in(**"你好"**);  s.in(**"我好"**);  s.in(34);   s.show();  *//当栈中不为空就一直取数据* **while**( !s.isEmpty()){  System.***out***.print(s.out()+**" "**);  }  }  } *//模拟栈内存的存储特点:先进去的后出* **class** Stack{  *//用LinkedList作为存储媒介* **private** LinkedList **list**= **new** LinkedList();  *//模拟进栈* **public void** in(Object obj){  **list**.addLast(obj);*//后进的在顶上 在尾部追加* }  *//模拟弹栈 出栈 从最后进去的开始删除* **public** Object out(){  **return list**.removeLast();*//取出最后的元素,并且移除* }   *//判盘栈是否为空* **public boolean** isEmpty(){  **if**(**list**.size() ==0){  **return true**;  }**else**{  **return false**;  }  }  *//展示栈里面的信息* **public void** show(){  System.***out***.println(**list**);  } } |

### ●集合框架(泛型的由来)(了解)

A:案例演示

|  |
| --- |
| **package** com.sxt.list;  **import** java.util.ArrayList; **import** java.util.Iterator;  **public class** Demo3 {  **public static void** main(String[] args) {  ArrayList list =**new** ArrayList();  list.add(**"a"**);  list.add(**"abc"**);  list.add(34);  list.add(**"bcd"**);  Iterator it = list.iterator();  **while**(it.hasNext()){  Object obj = it.next();  String s = (String)obj;  System.***out***.println(s);  }  } } |
| 会报异常:java.lang.ClassCastException,是因为集合中有不同的数据类型,不能强转成String,在开发中这种情况会经常出现  ArrayList<Person> list=new ArrayList<Person>(); list.add(new Person("张三",23)); list.add(new Person("李四",24)); Iterator<Person> it=list.iterator(); while (it.hasNext()){  //System.out.println(it.next());  //张三 24 it.next()只能调用一次 调用多次会将指针向后移动多次  //System.out.println(it.next().getName()+"---"+it.next().getAge());  Person p=it.next();  System.*out*.println(p.getName()+"---"+p.getAge()); } |
| **package** com.sxt.list;  **import** java.util.ArrayList; **import** java.util.Iterator;  **public class** Demo3 {  **public static void** main(String[] args) {  *//<String>泛型,加上之后这个集合只能装指定泛型的数据类型,目的让运行过程中可能会出现异常的问题,提前到编译阶段 // ArrayList<String> list =new ArrayList<String>();* ArrayList<String> list =**new** ArrayList<>(); *//在jdk1.7版本之后支持,后面只写<>跟随前面的泛型中的数据类型(也称为菱形泛型)* list.add(**"a"**);//添加进去的数据类型保持原有的类型String  list.add(**"abc"**); *// list.add(34);* list.add(**"bcd"**);  Iterator it = list.iterator();  **while**(it.hasNext()){  Object obj = it.next();  String s = (String)obj;  System.***out***.println(s);  }  } } |
| **package** com.sxt.list;  **import** java.util.ArrayList;  **public class** Demo3 {  **public static void** main(String[] args) {  ArrayList<Character> list= **new** ArrayList<>();  list.add(**'d'**);//添加进去的数据类型保持原有的类型Character  list.add(**'你'**);  list.add(**'他'**);  list.add(**'我'**);  **for** (**int** i = 0; i <list.size() ; i++) {  Character c = list.get(i); *//不用再强转,存取取数据类型没有发生自动提升为Object* System.***out***.println(c);  }   } } |

泛型的由来:通过Object转型问题引入

Object类型可以接收任意的对象类型，但是在实际的使用中，会有类型转换的问题。也就存在隐患， 所以Java提供了泛型来解决这个安全问题。

public class Person {  
 private String name;  
 private int age;

class Student extends Person

class Worker extends Person

public class Tool {  
 private Object obj;

Tool t=new Tool(); //创建工具类对象  
t.setObj(new Student("张三",23));  
Worker w=(Worker) t.getObj();  
System.*out*.println(w);

类型转换异常：你set是学生 get是Worker

泛型里面不要写Object，能用，但是泛型没有意义

### ●集合框架(泛型类/泛型方法的概述和使用)(了解)

定义泛型

泛型字符可以是任何标识符，一般采用几个标记：E、T、K、V、N、？。

|  |  |  |
| --- | --- | --- |
| 泛型标记 对应单词 说明 | | |
| E | Element | 在容器中使用，表示容器中的元素 |
| T | Type | 表示普通的 JAVA 类 |

|  |  |  |
| --- | --- | --- |
| K | Key | 表示键，例如：Map 中的键 Key |
| V | Value | 表示值 |
| N | Number | 表示数值类型 |
| ？ |  | 表示不确定的 JAVA 类型 |

### ●集合框架(泛型类的概述和使用)(了解)

A: 泛型类就是把泛型定义在类上，用户使用该类的时候，才把类型明确下来。泛型类的具体使用方法是在类的名称后添加一个或多个类型参数声明，如：<T>、<T,K,V>

B:定义格式

|  |
| --- |
| **public class** 类名<泛型表示符号> {  public <泛型表示符号> void setMsg(泛型表示符号 msg){  }  public <泛型表示符号> 泛型表示符号 setMsg(泛型表示符号 msg){  }  } |

public class Tool<T> {  
 private T t;  
  
 public T getObj() {  
 return t;  
 }  
  
 public void setObj(T t) {  
 this.t = t;  
 }  
}

Tool<Student> t=new Tool<>(); //创建工具类对象  
t.setObj(new Student("张三",23));  
Worker w=(Worker) t.getObj();报错了

System.*out*.println(t.getObj().getName());

C:案例演示

泛型类的使用

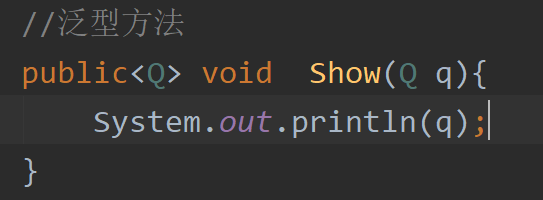
|  |
| --- |
| **public class** Demo2 {  **public static void** main(String[] args) {  Animal<String> a= **new** Animal();  a.setMsg(**"小黄"**);  System.***out***.println(a.getMsg());  } } **class** Animal<T>{  **private** T **msg**;  **public void** setMsg(T msg){  **this**.**msg** = msg;  }  **public** T getMsg(){  **return msg**;  } } |

//泛型方法  
public void Show(T t){  
 System.*out*.println(t);  
}

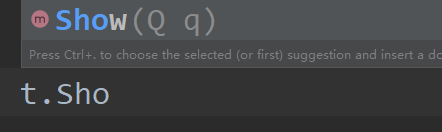
Tool<String> t=new Tool<>();  
t.Show("abc");

//泛型方法  
public void Show(Q q){  
 System.*out*.println(q); 会报错  
}

修改 方法拥有了自己的泛型

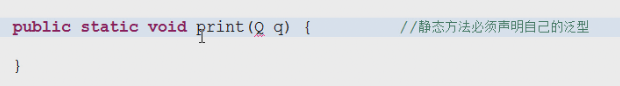


调用：

具体给类型

t.Show(true) 打印就是true

方法泛型最好与类的泛型一致 如果不一致需要在方法上声明该泛型



这里的Q和类中间的Q不一样 一个是在类加载的时候初始化赋值 一个是在调用静态方法的时候初始化赋值



### ●集合框架(泛型接口的概述和使用)(了解)

A:泛型接口概述

把泛型定义在接口上

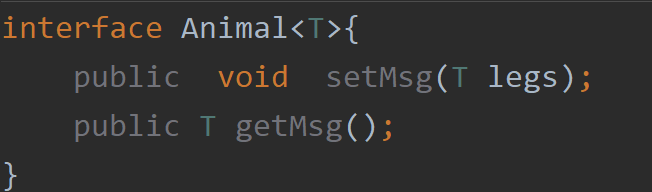
B:定义格式

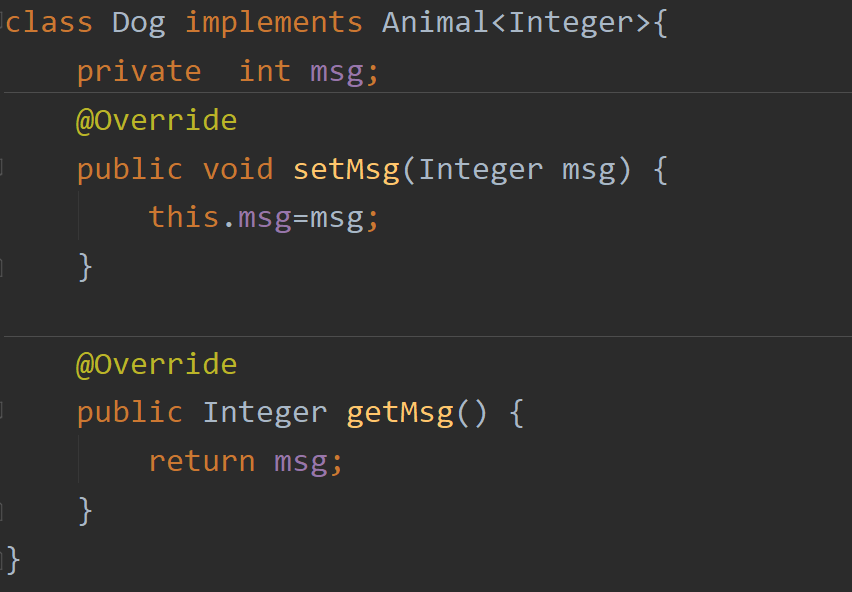
|  |
| --- |
| **public interface** 接口名<泛型表示符号> {  } |

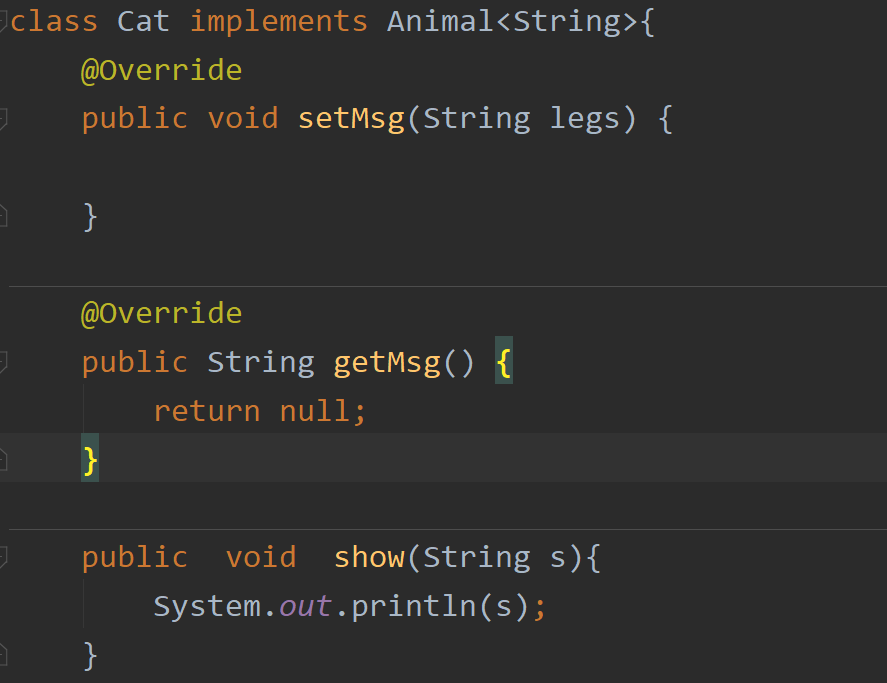
C:案例演示

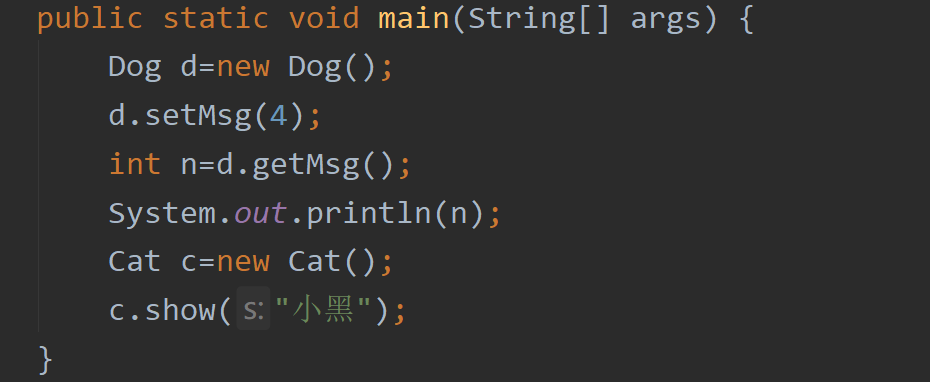
泛型接口和泛型类的声明方式一致。泛型接口的具体类型需要在实现类中进行声明。

|  |
| --- |
| **public class** Demo2 {  **public static void** main(String[] args) {  Dog d = **new** Dog();  d.setMsg(4);  **int** n =d.getMsg();   System.***out***.println(n);  } } **interface** Animal<T>{  **public void** setMsg(T legs);  **public** T getMsg(); } **class** Dog **implements** Animal<Integer>{  **private int legs**;  @Override  **public void** setMsg(Integer legs) {  **this**.**legs** = legs;  }  @Override  **public** Integer getMsg() {  **return legs**;  } } |









**●集合框架(通配符)(掌握)**

无界通配符

“?”表示类型通配符，用于代替具体的类型。它只能在“<>”中使用。可以解决当具体类型不确定的问题。

当右边的泛型不确定时，左边可以指定为？

语法结构

|  |
| --- |
| **public void** ShowMsg(Generic<?> generic){  } |

案例:

|  |
| --- |
| **public class** Generic<T> {  **private** T **msg**;   **public void** setMsg(T msg){  **this**.**msg** = msg;  }  **public** T getMsg(){  **return this**.**msg**;  } } |
| **public class** Demo1 {  **public static void** main(String[] args) {   Generic<String> g = **new** Generic<>();  g.setMsg(**"今天是个好日子"**);  ShowMsg sm = **new** ShowMsg();  sm.show(g);  }  } **class** ShowMsg {  **public void** show(Generic<?> generic){ *//可以传入的具体类型不确定* System.***out***.println(generic.getMsg());  } } |

**●集合框架(统配符的上限限定)(掌握)**

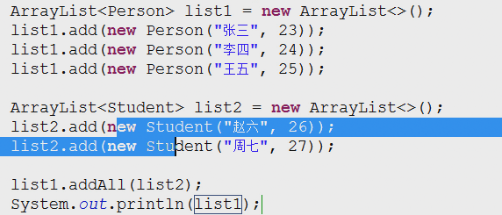
上限限定表示通配符的类型是 T 类以及 T 类的子类或者 T 接口以及 T 接口的子接口。该方式同样适用于与泛型的上限限定。

|  |
| --- |
| **public void** ShowMsg(Generic<? **extends Person**> generic){  } |

建立一个Person类 提供toString()

学生继承Person

在main()中添加人 学生



打印出是Person中的方法 Student必须继承Person 子类提升为父类

反着放不让放 父类不可以提升子类

子类型？ 继承父类 向下限

案例:

|  |
| --- |
| **public class** Generic<T> {  **private** T **msg**;   **public void** setMsg(T msg){  **this**.**msg** = msg;  }  **public** T getMsg(){  **return this**.**msg**;  } } |
| **public class** Person {  **private** String **name**;  **private int age**;   **public** Person() {  }   **public** Person(String name, **int** age) {  **this**.**name** = name;  **this**.**age** = age;  }   @Override  **public** String toString() {  **return "Person{"** +  **"name='"** + **name** + **'\''** +  **", age="** + **age** +  **'}'**;  } } |
| **class** Student **extends** Person{  **public** Student(){  **super**();  }  **public** Student(String name, **int** age) {  **super**(name,age);  } } |
| **public class** Demo1 {  **public static void** main(String[] args) {   Generic<Student> g = **new** Generic<>();  Student stu = **new** Student(**"静静"**,19);  g.setMsg(stu);  ShowMsg sm = **new** ShowMsg();  sm.show(g);  }  } **class** ShowMsg {  *//可以传入Person类型,或其子类(类型<=*Person*)*  **public void** show(Generic<? **extends** Person> generic){ System.***out***.println(generic.getMsg());  } } |

**●集合框架(通配符的下限限定)(掌握)**

下限限定表示通配符的类型是 T 类以及 T 类的父类或者 T 接口以及 T 接口的父接口。

注意：该方法不适用泛型类。

|  |
| --- |
| **public void** ShowMsg(Generic<? **super** Student> generic){  } |

示例

|  |
| --- |
| **public class** Generic<T> {  **private** T **msg**;   **public void** setMsg(T msg){  **this**.**msg** = msg;  }  **public** T getMsg(){  **return this**.**msg**;  } }  **public class** Person {  **private** String **name**;  **private int age**;   **public** Person() {  }   **public** Person(String name, **int** age) {  **this**.**name** = name;  **this**.**age** = age;  }   @Override  **public** String toString() {  **return "Person{"** +  **"name='"** + **name** + **'\''** +  **", age="** + **age** +  **'}'**;  } } |
| **class** Student **extends** Person{  **public** Student(){  **super**();  }  **public** Student(String name, **int** age) {  **super**(name,age);  } } |
| **public class** Demo1 {  **public static void** main(String[] args) {  *// Generic<Student> g = new Generic<>(); // g.setMsg(new Student("李四",20));* Generic<Person> g2 = **new** Generic<>();  g2.setMsg(**new** Person(**"李四"**,20));  ShowMsg sm = **new** ShowMsg();  sm.show(g2);  }  }  **class** ShowMsg {  *//可以传入Person类型,或其父类(类型>=Student)*  **public void** show(Generic<? **super** Student> generic){  System.***out***.println(generic.getMsg());  } } |

**●泛型总结(掌握)**

泛型主要用于编译阶段，编译后生成的字节码 class 文件不包含泛型中的类型信息。 类型参数在编译后会被替换成 **Object**，运行时虚拟机并不知道泛型。因此，使用泛型时，如下几种情况是错误的：

1. 基本类型不能用于泛型。

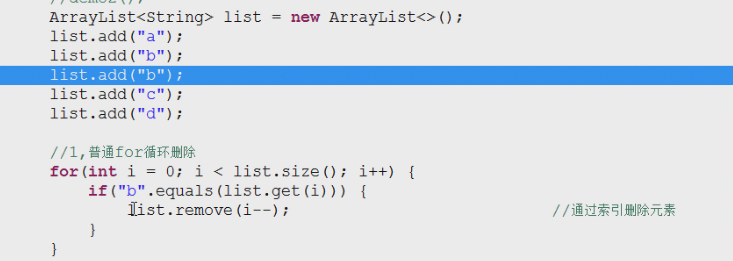
Test<int> t; 这样写法是错误，我们可以使用对应的包装类；Test<Integer> t ;

1. 不能通过类型参数创建对象。

T elm = new T(); 运行时类型参数 T 会被替换成 Object，无法创建 T 类型的对象，容易引起误解，所以在 Java 中不支持这种写法。

### ●集合框架(三种迭代的能否删除)(掌握)





|  |
| --- |
| **package** com.sxt.list;  **import** java.util.ArrayList; **import** java.util.Iterator; **import** java.util.ListIterator;  **public class** Demo3 {  **public static void** main(String[] args) {  ArrayList<Teacher> list= **new** ArrayList<>();  list.add(**new** Teacher(**"Niko"**,**"数学班"**));  list.add(**new** Teacher(**"Niko"**,**"体育班"**));  list.add(**new** Teacher(**"陈老师"**,**"语文班"**));  *//如果集合中有李老师,就移除李老师* ListIterator<Teacher> lit = list.listIterator();  **while**(lit.hasNext()){  Teacher t = lit.next();  **if**(**"NiKo"**.equals(t.getName())){  lit.remove();*//用迭代器移除集合中的对象,会把除以的操作同步到集合中* }  }  System.***out***.println(list);    } } **class** Teacher{  **private** String **name**;  **private** String **classType**;   **public** Teacher() {  }   **public** Teacher(String name, String classType) {  **this**.**name** = name;  **this**.**classType** = classType;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public** String getClassType() {  **return classType**;  }   **public void** setClassType(String classType) {  **this**.**classType** = classType;  }   @Override  **public** String toString() {  **return "Teacher{"** +  **"name='"** + **name** + **'\''** +  **", classType='"** + **classType** + **'\''** +  **'}'**;  } } |
| list集合遍历有三种方法:for,增强for循环,迭代器里面,在遍历过程中如果要增加或移除集合中的元素,必须使用ListIterator迭代器,  for,增强for循环:会发生并发修改异常 |

### ●集合框架(可变参数的概述和使用)(掌握)

A:可变参数概述

定义方法的时候不知道该定义多少个参数

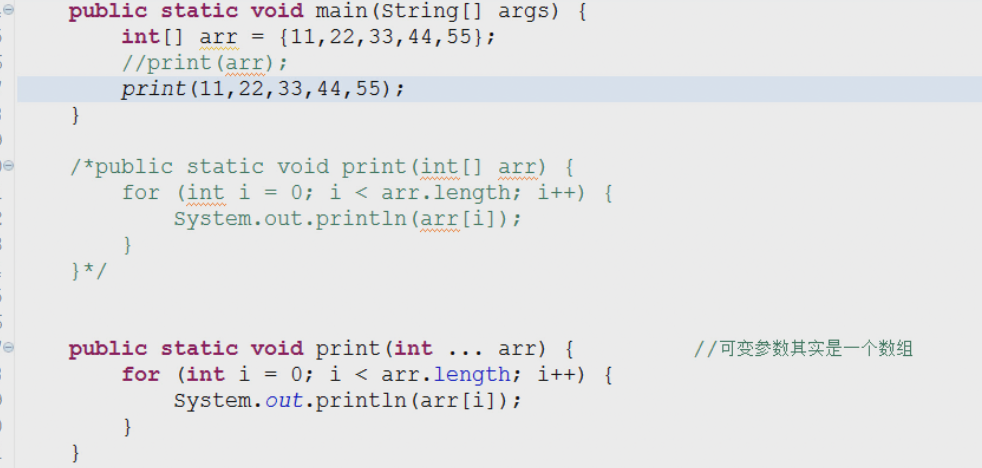
B:格式

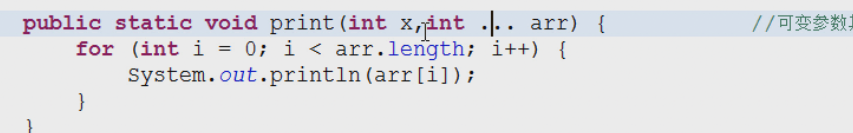
修饰符 返回值类型 方法名(数据类型… 变量名){}

C:注意事项：

这里的变量其实是一个数组

如果一个方法有可变参数，并且有多个参数，那么，可变参数肯定是最后一个





22 33 44 55 11赋值给x了 将参数对调 报错

|  |
| --- |
| **package** com.sxt.list;  **public class** Demo5 {  **public static void** main(String[] args) { *// add(12,20,30,40,50);//可以传入n个参数  add2*(10,30,50,70,79,30);  }  **public static void** add(**int** ...a){ *//int ...a是可变形式参数,相当于是个数组* **for** (**int** i = 0; i <a.**length** ; i++) {  System.***out***.println(a[i]);  }  }  **public static void** add2(**int** b,**int** ...a){ *//int ...a是可变形式参数一定在最后一个* System.***out***.println(**"b="**+b);  **for** (**int** i = 0; i <a.**length** ; i++) {  System.***out***.println(a[i]);  }  } } |
| 封装数据库JDBC里面增删改查会使用这个可变参数 |

### ●集合框架(Arrays工具类的asList()方法的使用)(掌握)

A:案例演示

Arrays工具类的[asList](mk:@MSITStore:C:\\工作\\备课\\API文档\\JDK%201.6%20API.chm::/java/util/../../java/util/Arrays.html" \l "asList(T...))(T... a) 方法的使用

结论：数组转结合，数组的类型必须是引用数据类型

开发中数组转集合：把数组遍历之后，然后通过add方法把每个元素一一加入集合

|  |
| --- |
| **package** com.sxt.list;  **import** java.util.ArrayList; **import** java.util.Arrays; **import** java.util.List;  **public class** Demo6 {  **public static void** main(String[] args) {  List<Integer> li= Arrays.*asList*(3,4,5,6,7,78,8);  **for** (**int** i = 0; i <li.size() ; i++) {  System.***out***.println(li.get(i));  } *// li.add(10); //通过Arrays.asList方法转成的集合是一个整体,不能往里面添加或删除新的元素 // li.remove(4); //UnsupportedOperationException* System.***out***.println(li);    基本数据类型的数组转换成集合，会将整个数组当成一个对象存储到集合中 数组中存储的是基本数据类型 集合只能添加引用数据类型 所以将arr当成一个对象进行存储 打印出来为地址值。  想将数组转换为集合 数组必须是引用数据类型    */\*  目的是:使用集合中一些比较好用的方法  \*/* System.***out***.println(li.get(0));   *//开发中把一个数组转成集合* **int**[]arr = {3,4,5,6,7,78,8};  ArrayList<Integer> list= **new** ArrayList<>();  **for**(**int** i:arr){  list.add(i);  }  System.***out***.println(**"list:"**+list);  list.add(10);  list.remove(0);  System.***out***.println(**"list:"**+list);  } } |

Collection中toArray(T[] a)泛型版的集合转数组

|  |
| --- |
| **package** com.sxt.list;  **import** java.util.ArrayList; **import** java.util.Arrays; **import** java.util.Collection;  **public class** Demo7 {  **public static void** main(String[] args) {  ArrayList<String> list= **new** ArrayList<>();  list.add(**"长沙"**);  list.add(**"中部"**);  list.add(**"老二"**);  list.add(**"城市"**);  *//数组转集合,ArrayList间接实现了Collection接口中的toArray(T[] a)方法* String[] arr = list.toArray(**new** String[0]); *//用泛型的方式把集合转成数组,传入数组匿名对象,如果传入的数组长度大于集合中元素的个数,那么不够用默认值替代*  *当集合转数组时，数组长度如果是小于等于集合的长度 转换后的数组长度等于集合的长度*  *如果数组的长度大于size,分配的长度就和你指定的长度一样*  **for** (String s:arr) {  System.***out***.println(s);  }  System.***out***.println(arr.**length**);  arr[0] = **"武汉"**;  System.***out***.println(Arrays.*toString*(arr));  *//集合转成的数组,可以当做普通的数组操作* } } |

### ●集合框架(集合嵌套之ArrayList嵌套ArrayList)(了解)

A:案例演示

集合嵌套之ArrayList嵌套ArrayList(了解)

|  |
| --- |
| **package** com.sxt.list;  **import** java.util.ArrayList; **import** java.util.Arrays;  **public class** Demo8 {  **public static void** main(String[] args) {  ArrayList<ArrayList<Person>> list = **new** ArrayList<>(); *//节目单* ArrayList<Person> one = **new** ArrayList<>();*//唱歌组* ArrayList<Person> two = **new** ArrayList<>();*//小品组* one.add(**new** Person(**"孙俪"**,30));  one.add(**new** Person(**"李健"**,31));  one.add(**new** Person(**"张学友"**,33));   two.add(**new** Person(**"岳云鹏"**,36));  two.add(**new** Person(**"于谦"**,40));  two.add(**new** Person(**"郭德纲"**,39));   list.add(one);  list.add(two);   **for**(ArrayList<Person> arr:list){ 拿到每个集合  **for**(Person p:arr){ 拿到集合中每个元素  System.***out***.println(p.getName()+**"--"**+p.getAge());  }  }  } } **class** Person{  **private** String **name**;  **private int age**;   **public** Person() {  }   **public** Person(String name, **int** age) {  **this**.**name** = name;  **this**.**age** = age;  }   **public** String getName() {  **return name**;  }   **public void** setName(String name) {  **this**.**name** = name;  }   **public int** getAge() {  **return age**;  }   **public void** setAge(**int** age) {  **this**.**age** = age;  } } |

List接口下的集合:重点掌握集合的遍历,遍历过程中添加或移除元素(注意并发修改异常)