第7天 集合

今日学习内容

* Collection集合
* Iterator迭代器
* 增强for循环
* 泛型

今日学习目标

* 能够说出集合与数组的区别
* 说出Collection集合的常用功能
* 能够使用迭代器对集合进行取元素
* 能够说出集合的使用细节
* 能够使用集合存储自定义类型
* 能够使用foreach循环遍历集合
* 能够使用泛型定义集合对象
* 能够阐述泛型通配符的作用

# 集合

## 集合

### 集合介绍

前面的学习，我们知道数据多了，使用数组存放。而且数组中存放的都是基本类型的数据，并且数组是定长的。

当在程序中创建的对象比较多的时候，需要对这些对象进行统一的管理和操作，那么首先我们就需要把这些对象存储起来。使用数组是可以存放对象的，我们可以定义对象数组来存放，但是数组这个容器存放对象，要对其中的对象进行更复杂操作时，数据就显的很麻烦。那怎么办呢？

Java中给我们提供了另外一类容器，专门用来存放对象，这个容器就是我们要学习的集合。

集合和数组既然都是容器，它们有啥区别呢？

数组的长度是固定的。集合的长度是可变的。

数组中存储的是同一类型的元素，可以存储基本数据类型值。

集合存储的都是对象。而且对象的类型可以不一致。ArrayList<Object>

集合貌似看起来比较强大，它啥时用呢？

当对象多的时候，先进行存储。

### 集合框架的由来

集合本身是一个工具，它存放在java.util包中。

JDK最早的1.0版本中。提供的集合容器很少Vector。升级到1.2版，为了更多的需求，出现了集合框架。有了更多的容器(为什么提供那么多集合容器呢?)。可以完成不同的需求。

这些容器怎么区分？区分的方式：每一个容器的数据结构(数据存储一种方式)不一样。

例如:水缸和衣柜,饮料瓶等等.

不同的容器进行不断的向上抽取，最后形成了一个集合框架，这个框架就是Collection接口(一会可以看一下API)。在Collection接口定义着集合框架中最最最共性的内容。

在学习时：我们需要看最顶层怎么用， 创建底层对象即可。因为底层继承了父类中的所有功能。

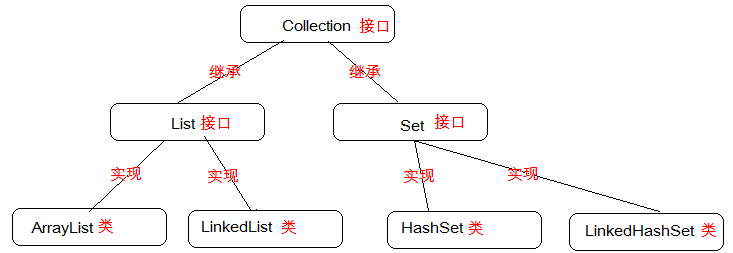
### 接口的描述

既然Collection接口是集合中的顶层接口，那么它中定义的所有功能子类都可以使用。查阅API中描述的Collection接口。Collection 层次结构 中的根接口。Collection 用来存储一组对象，这些对象也称为 collection 的元素。一些 collection 允许有重复的元素，而另一些则不允许。

一些 collection 是有序的，而另一些则是无序的。

其实我们在使用ArrayList类时，该类已经把所有抽象方法进行了重写。那么，实现Collection接口的所有子类都会进行方法重写。

* Collecton接口常用的子接口有：List接口(列表)、Set接口(集)
* List接口常用的子类有：ArrayList类、LinkedList类
* Set接口常用的子类有：HashSet类、LinkedHashSet类



继续查阅API，发现Collection接口中很多集合的操作方法，那么这些方法都具体能做什么呢？

### Collection基本方法了解

## Collection接口的基本方法



这里我们不关心具体创建的Collection中的那个子类对象，这里重点演示的是Collection接口中的方法

Collection<String> coll = new ArrayList<String>();

//1,往集合中添加对象元素。add(Object);

coll.add("itcast1");

coll.add("itcast2");

coll.add("itcast3");

//2,删除。

coll.remove("itcast2");

//3,判断是否包含。

System.*out*.println(coll.contains("itcast11"));

//4,清除。

coll.clear();

//把集合打印一下。

System.*out*.println(coll);//[itcast1, itcast2, itcast3]

# Iterator迭代器

## Iterator迭代器概述

java中提供了很多个集合，它们在存储元素时，采用的存储方式不同。我们要取出这些集合中的元素，可通过一种通用的获取方式来完成。

Collection集合元素的通用获取方式：在取元素之前先要判断集合中有没有元素，如果有，就把这个元素取出来，继续在判断，如果还有就再取出出来。一直把集合中的所有元素全部取出。这种取出方式专业术语称为迭代。

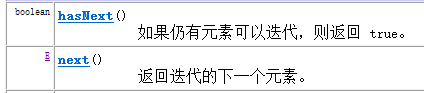
我们来画图描述一下迭代器的作用.

## Iterator迭代方式的代码体现

在Collection接口描述了一个抽象方法iterator方法，所有Collection子类都实现了这个方法，并且有自己的迭代形式。



集合中把这种取元素的方式描述在Iterator接口中。Iterator接口的常用方法如下：



* hasNext（）方法：用来判断集合中是否有下一个元素可以迭代。如果返回true,说明可以迭代。
* next（）方法：用来返回迭代的下一个元素，并把指针向后移动一位。

进行代码演示：

//1，创建集合对象。

Collection<String> coll = **new** ArrayList<String>();

coll.add("abc1");

coll.add("abc2");

coll.add("abc3");

coll.add("abc4");

//2.获取容器的迭代器对象。通过iterator方法。

Iterator it = coll.iterator();

//3,使用具体的迭代器对象获取集合中的元素。参阅迭代器的方法

while(it.hasNext()){

System.out.println(it.next());

}

/\*

迭代器for循环的形式的使用

**for** (Iterator it = coll.iterator(); it.hasNext(); ) {

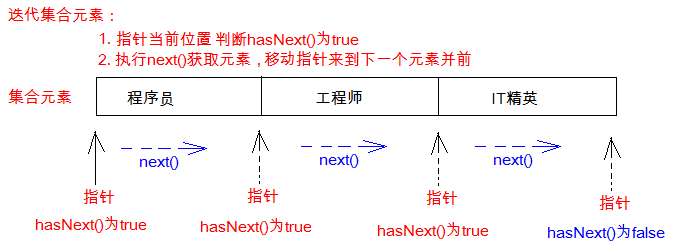
System.*out*.println(it.next());

}

\*/

注意：在进行集合元素取出时，如果集合中已经没有元素了，还继续使用迭代器的next方法，将会发生java.util.NoSuchElementException没有集合元素的错误。

**迭代集合元素图解：**



下边分别介绍以上内容：

1. Collection接口的iterator

方法声明为：

Iterator<集合中数据类型> iterator()

用来返回专属于该集合对象的迭代器对象(Iterator的子类对象)。

1. Iterator接口

该接口规定了迭代集合所需要的方法

1. Iterator接口的两个方法：hasNext与next方法

Iterator规定了两个方法，集合对象产生的迭代器对象正是通过这两个方法帮助集合进行迭代工作的。

调用迭代器的hasNext方法判断是否有下一个元素

调用迭代器的next获取下一个元素

## 并发修改异常

举例:

使用Iterator对象或者增强for循环遍历集合,如果出现”itcast”字符串,那么向集合中添加一个大写的”ITCAST”字符串

迭代的常规用法中我们要尽量避免在迭代过程中为集合添加/删除数据。否则会报错，原因是Java抛出了并发修改异常。

迭代过程中并发修改异常的原因为:

迭代器中”记忆”的集合长度与集合中实际长度不同，而导致出现索引与实际元素不符甚至无限循环的情况发生。

所以在使用Iterator时，避免类似操作，for循环底层为迭代器实现，所以也需要避免类似操作。

有些迭代器避免了这样的问题，如ListIterator，但该类并不通用也不常用，实际开发中很少使用，只需要简单了解。

java中提供了很多个集合，它们在存储元素时，采用的存储方式不同。我们要取出这些集合中的元素，可通过一种通用的获取方式来完成。

## 增强for循环

增强for循环是JDK1.5以后出来的一个高级for循环，专门用来遍历数组和集合的。它的内部原理其实是个Iterator迭代器，所以在遍历的过程中，不能对集合中的元素进行增删操作。

格式：

for(元素的数据类型 变量 : Collection集合or数组){

}

它用于遍历Collection和数组。通常只进行遍历元素，不要在遍历的过程中对集合元素进行增删操作。

练习一：遍历数组

int[] arr = new int[]{11,22,33};

for (int n : arr) {//变量n代表被遍历到的数组元素

System.out.println(n);

}

练习二:遍历集合

Collection<String> coll = new ArrayList<String>();

coll.add("itcast1");

coll.add("itcast2");

coll.add("itcast3");

coll.add("itcast4");

for(String str : coll){//变量Str代表被遍历到的集合元素

System.*out*.println(str);

}

增强for循环和老式的for循环有什么区别？

注意：新for循环必须有被遍历的目标。目标只能是Collection或者是数组。

建议：遍历数组时，如果仅为遍历，可以使用增强for如果要对数组的元素进行 操作，使用老式for循环可以通过角标操作。

# 泛型

## 泛型概述

泛型:泛泛的类型,就是一种不确定的类型(JDK1.5的一个新特性)

基本体现: <E>这就是泛型,此处的E是什么数据类型?

泛型用来灵活地将数据类型应用到不同的类、方法、接口当中。将数据类型作为参数传递。

泛型是数据类型的一部分，我们将类名与泛型合并一起看做数据类型。

泛型的定义：定义泛型可以在类中预支地使用未知的类型。

泛型的使用：一般在创建对象时，将未知的类型确定具体的类型。当没有指定泛型时，默认类型为Object类型。

## 使用泛型的好处

* 将运行时期的ClassCastException，转移到了编译时期变成了编译失败。
* 避免了类型强转的麻烦。

演示下列代码：

**publicclass** GenericDemo {

/\*

\* 使用集合对象存储并遍历

\* 不使用泛型 <> 不写

\*/

public static void demo01(){

//创建集合对象,不使用泛型

//没有明确集合存储的数据类型是什么, 什么都能存 Object

ArrayList array = new ArrayList();

array.add("abc");

array.add(123);

Iterator it = array.iterator();

while(it.hasNext()){

//迭代器的方法next()返回值是什么 Object

//System.out.println(it.next());

String s = (String)it.next();

System.out.println(s);

}

}

/\*

\*使用泛型

\*/

**publicstaticvoid** demo02() {

Collection<String> list = **new** ArrayList<String>();

list.add("abc");

list.add("itcast");

//list.add(5);//当集合明确类型后，存放类型不一致就会编译报错

//集合已经明确具体存放的元素类型，那么在使用迭代器的时候，迭代器也同样会知道具体遍历元素类型

Iterator<String> it = list.iterator();

**while**(it.hasNext()){

String str = it.next();

//当使用Iterator<String>控制元素类型后，就不需要强转了。获取到的元素直接就是String类型

System.*out*.println(str.length());

}

}

}

## 泛型的定义与使用

我们在集合中会大量使用到泛型，这里来完整地学习泛型知识。

泛型，用来灵活地将数据类型应用到不同的类、方法、接口当中。将数据类型作为参数进行传递。

### 泛型中E的含义

我们以ArrayList<E>为例:

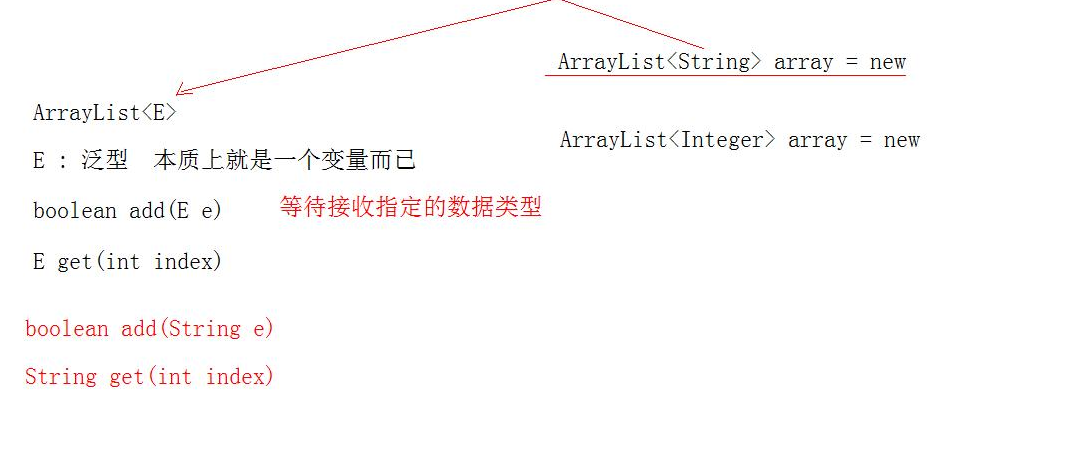
类名:ArrayList<E>

方法:

boolean add(E e);

E get(int index)

画图说明:



### 含有泛型的类:泛型类

定义格式：修饰符 class 类名<代表泛型的变量> { }

* + 例如，API中的ArrayList集合：

class ArrayList**<E>**{

public boolean **add**(**E** e){ }

public **Eget**(int index){ }

}

使用格式：创建对象时，确定泛型的类型

* + 例如，ArrayList<String> list = new ArrayList<String>();

此时，变量E的值就是String类型

class ArrayList**<String>**{

public boolean **add**(**String** e){ }

public **Stringget**(int index){ }

}

* + 例如，ArrayList<Integer> list = new ArrayList<Integer>();

此时，变量E的值就是Integer类型

class ArrayList**<Integer>**{

public boolean **add**(**Integer** e){ }

public **Integerget**(int index){ }

}

举例自定义泛型类

**publicclass** GenericClass<E>{//自定义的类中,可以写<>泛型

//E 表示未知的数据类型 调用者创建对象的时候,才能明确数据类型

**private** E e;

**publicvoid** setE(E e){

**this**.e = e;

}

**public** E getE(){

**return**e;

}

}

使用:

**publicclass** GenericClassTest {

**publicstaticvoid** main(String[] args) {

//对自定义的泛型类,进行测试

GenericClass<Integer> g = **new** GenericClass<Integer>();

//E传递什么类型就是什么类型

g.setE(100);

Integer i = g.getE();

System.*out*.println(i);

}

}

### 含有泛型的方法

定义格式：修饰符 <代表泛型的变量> 返回值类型 方法名(参数){ }

例如，

**publicclass** GenericMethod <E>{

**Public void** show1(E e){

System.*out*.println(e);

}

**public** <T> **void** show2(T t){//自定义泛型的方法

//自己写一个方法,方法中的数据类型,采用<>泛型

//如果方法中的泛型,和类上的泛型不同

// 在方法返回值前加入<>

System.*out*.println(t);

}

}

使用格式：调用方法时，确定泛型的类型

**Public class** GenericMethodTest {

**Public static void** main(String[] args) {

GenericMethod<Double> g = **new** GenericMethod<Double>();

g.show(1.1);

g.function(1.2F);//传递什么类型就是什么类型

}

}

### 含有泛型的接口

定义格式：修饰符 interface接口名<代表泛型的变量> { }

* + 例如，

**publicinterface** Inter <E>{

**publicabstractvoid** show(E e);

}

使用格式：

1、实现接口时,确定泛型的类型

* + 例如

**publicclass** InterImpl **implements** Inter<Integer>{

**publicvoid** show(Integer i){

System.*out*.println(i);

}

}

此时，变量E的值就是Integer类型。

2、实现接口,不指定泛型的类型，直到创建对象时，确定泛型的类型

* + 例如

InterImpl<String> imp= new InterImpl<String>();

此时，变量E的值就是**String**类型。

**publicclass** InterImpl<E>**implements** Inter<E>{

**publicvoid** show(E e){

System.*out*.println(e);

}

}

## 泛型通配符

当使用泛型类或者接口时，传递的数据中，泛型类型不确定，可以通过通配符<?>表示。但是一旦使用泛型的通配符后，只能使用Object类中的共性方法，集合中元素自身方法无法使用。

定义：(查看ArrayList的构造方法或者addAll方法)无法在类中使用

使用：调用方法时可以给予任意类型。参照Arraylist的构造方法或者addAll方法

? extends E代表只要是E类型的子类即可

? super E代表只要是E类型的父类即可

/\*

\* 泛型通配符?,代表任意的数据类型

\*

\* 定义：(查看ArrayList的addAll)无法在类中使用

\*

\* 使用：调用方法时可以给予任意类型。参照Arraylist的add方法

\* public boolean addAll(Collection<? extends E> c)

\* 为了便于?的理解,我们将以上方法重写为public boolean addAll(ArrayList<? extends E> c)

\*

\* 该方法的意思:集合对象A调用addAll方法,将一个集合对象B作为参数,即向集合A中添加集合B中的元素

\*

\* ? extends E代表只要是E类型的子类即可

\* ? super E代表只要是E类型的父类即可

\*/

**publicclass** Demo01 {

**publicstaticvoid** main(String[] args) {

//定义集合b,包含3个元素

ArrayList<String> listB = **new** ArrayList<String>();

listB.add("Jack");

listB.add("Rose");

listB.add("Trump");

//使用集合b创建集合a

ArrayList<Object> listA = **new** ArrayList<Object>();

listA.add("Obama");

listA.addAll(listB);

//观察集合A

System.*out*.println(listA);

}

# 集合综合案例

## 案例介绍

按照斗地主的规则，完成洗牌发牌的动作。

具体规则：

使用54张牌

打乱顺序

三个玩家参与游戏，三人交替摸牌，每人17张牌，最后三张留作底牌。

## 案例需求分析

* 准备牌：

牌可以设计为一个ArrayList<String>,每个字符串为一张牌。

每张牌由花色数字两部分组成，我们可以使用花色集合与数字集合嵌套迭代完成每张牌的组装。

牌由Collections类的shuffle方法进行随机排序。

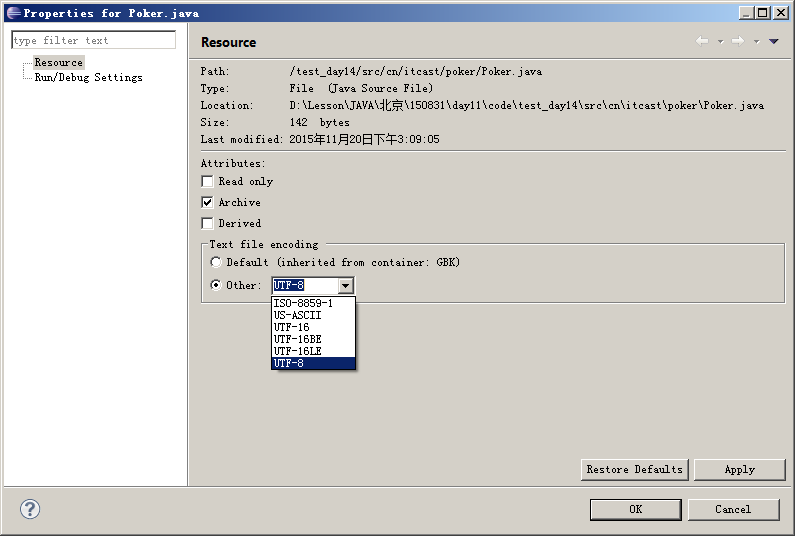
* 发牌：

将每个人以及底牌设计为ArrayList<String>,将最后3张牌直接存放于底牌，剩余牌通过对3取模依次发牌。

* 看牌：

直接打印每个集合。

## 实现代码步骤



修改文件编码由GBK修改为UTF-8，因为GBK没有我们要的梅花、方片、黑桃、红桃(♠♥♦♣)等字符。

publicclass Poker {

publicstaticvoid main(String[] args) {

//♠♥♦♣

//准备牌

ArrayList<String> poker = new ArrayList<String>();

//花色

ArrayList<String> color = new ArrayList<String>();

color.add("♠");

color.add("♥");

color.add("♦");

color.add("♣");

//数字

ArrayList<String> number = new ArrayList<String>();

for (int i = 2; i <= 10; i++) {

number.add(i+"");

}

number.add("J");

number.add("Q");

number.add("K");

number.add("A");

//完成新牌

for (String thisColor : color) {

for (String thisNumber : number) {

String thisCard = thisColor + thisNumber;

poker.add(thisCard);

}

}

poker.add("小☺");

poker.add("大☻");

//洗牌

Collections.*shuffle*(poker);

//发牌

//玩家1

ArrayList<String> player1 = new ArrayList<String>();

//玩家2

ArrayList<String> player2 = new ArrayList<String>();

//玩家3

ArrayList<String> player3 = new ArrayList<String>();

//底牌

ArrayList<String> secretCards = new ArrayList<String>();

for (int i = 0; i < poker.size(); i++) {

if(i>=51) {

//最后三张发给底牌

secretCards.add(poker.get(i));

}else {

//剩余牌通过对3取模依次摸牌

if(i%3==0) {

player1.add(poker.get(i));

}elseif(i%3==1) {

player2.add(poker.get(i));

}else {

player3.add(poker.get(i));

}

}

}

//看牌

System.*out*.println(player1);

System.*out*.println(player2);

System.*out*.println(player3);

System.*out*.println(secretCards);

}

}

* 最后发到三个人手中的牌是无序的，在明天学习完Map集合后，我们提供一个排序的解决方案。