* 今天学习目标

  * 能够理解泛型概述

  * 能够掌握泛型类

    * 箱

  * 能够掌握泛型方法

  * 能够掌握含有接口的泛型

  * 能够理解泛型通配符

  * 能够理解泛型上下限

    extends super

    * 抽出公共的方法

  * 能够使用集合完成综合案例

    * 斗地主

---

  * 能够理解泛型概述

  * 能够掌握泛型类

  * 能够掌握泛型方法

  * 能够掌握含有接口的泛型

  * 能够理解泛型通配符

  * 能够理解泛型上下限

  * 能够使用集合完成综合案例

---

  * 回顾

    * 自定义异常：extends Exception，RuntimeException，重写构造器

    * 日志：log4j（properties），log4j2（xml）

      * 级别，输出地，格式，配置文件

      * Logger(trace,debug,info,warn,error)

* Appender(ConsoleAppender,FileAppender,.....)

* Layout(PatternLayout,HTMLLayout,SImpleLayout)

* 集合：容器--->引用类型

* 与数组的区别：数组的长度不可变，集合的长度可变

* Collection,List(ArrayList,Vector,LinkedList),Set(HashSet(HashMap的key)，LinkedHashSet，TreeSet),Queue

* 多线程并发集合

* add,remove,contains,clear,isEmpty,addAll,removeAll,containsAll,toArray,size

* Iterable---> Iterator(hasNext,next)

* NoSuchElementException

* 增强型for（JDK1.5），迭代过程中，不可以删除或者增加元素

* ConcurrentModificationException

* 能够理解泛型概述

* 案例

```
* 没有加上泛型
public static void main(String[] args) {
        Collection container=new ArrayList();
        // add:东邪西毒南帝北丐中神通
        container.add("东邪");
        container.add(2);
        container.add("西毒");
        container.add("南帝");
        container.add("北丐");
        container.add("中神通");
        Iterator<String> ite = container.iterator();
        while(ite.hasNext()) {
            String value=ite.next();
            System.out.println(value);
        }
    }
结果并报异常：
东邪
```

```
Exception in thread "main" java.lang.ClassCastException:
    java.lang.Integer cannot be cast to java.lang.String
    at com.lg.test1.Test1.main(Test1.java:19)

* 泛型：
  Collection虽然可以存储各种对象，但实际上通常Collection只存储同一类型对象，例如Stri
  因此在JDK5之后，新增了泛型(Generic)语法，让我们在设计API时可以指定类或方法支持泛型，
  在编译时期进行语法检查，运行时擦除泛型。
* 什么是泛型？
    * 可以在类或方法中预支地使用未知的类型。
    * 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.
 * 添加泛型
  public static void main(String[] args) {
        Collection<String> container=new ArrayList<String>();
        // add:东邪西毒南帝北丐中神通
        container.add("东邪");
//      container.add(2);// 编译期就报错了
        container.add("西毒");
        container.add("南帝");
        container.add("北丐");
        container.add("中神通");
        Iterator<String> ite = container.iterator();
        while(ite.hasNext()) {
            String value=ite.next();
            System.out.println(value);
        }
    }
 * 泛型好处
    * 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
```

* 能够掌握泛型类

  * 泛型的意义在于代码的复用

    * 定义格式：

      * 修饰符 class 类名<代表泛型的变量> { }

      * 查看ArrayList源码

* 盒子案例

```
* 装黄金的盒子
public class Gold {
    @Override
    public String toString() {
        return "黄金";
    }

    public void lignt() {
        System.out.println("金闪闪...");
    }
}

public class GoldBox {
    private Gold gold;

    public Gold getGold() {
        return gold;
    }

    public void setGold(Gold gold) {
        this.gold = gold;
    }

}

public static void main(String[] args) {
        // 黄金的盒子
        GoldBox boxGold=new GoldBox();
        boxGold.setGold(new Gold());
        System.out.println(boxGold.getGold());
        boxGold.getGold().lignt();
    }
结果:
黄金
金闪闪...

* 装硬币的盒子
```

```
public class Coin {

    @Override
    public String toString() {
        return "硬币";
    }

    public void buy() {
        System.out.println("拿着一块买两包辣条...");
    }
}

public class CoinBox {
    private Coin coin;

    public Coin getCoin() {
        return coin;
    }

    public void setCoin(Coin coin) {
        this.coin = coin;
    }

}

public static void main(String[] args) {
        // 硬币的盒子
        CoinBox coinBox=new CoinBox();
        coinBox.setCoin(new Coin());
        System.out.println(coinBox.getCoin());
        coinBox.getCoin().buy();
    }
结果:
硬币
拿着一块买两包辣条...

思考：假如我要装其他的东西，是不是每次都要创建各种类型的箱子？
  * 用Object做个通用的箱子
public class ObjectBox {
    private Object boj;
```

```
78
79      public Object getBoj() {
80          return boj;
81      }
82
83      public void setBoj(Object boj) {
84          this.boj = boj;
85      }
86
87  }
88
89  public static void main(String[] args) {
90          // 黄金的盒子
91          ObjectBox boxGold=new ObjectBox();
92          boxGold.setBoj(new Gold());
93          Gold gold=(Gold) boxGold.getBoj();// 有可能异常
94          System.out.println(gold);
95          gold.lignt();
96          System.out.println("----------------------------------");
97          // 硬币的盒子
98          ObjectBox coinGold=new ObjectBox();
99          coinGold.setBoj(new Coin());
100         Coin coin=(Coin) coinGold.getBoj();
101         System.out.println(coin);
102         coin.buy();
103     }
104 结果:
105 黄金
106 金闪闪...
107 ----------------------------------
108 硬币
109 拿着一块买两包辣条...
110
111 思考:
112  * 使用Object作为箱子有什么问题?
113     * 类型强转比较麻烦
114     * 就像我们集合一样什么类型都可以装,在获取对象时候，容易出现ClassCastException,
115
116 使用泛型类
117 public class Box<T> {
```

```
    private T item;
    public T getItem() {
        return item;
    }
    public void setItem(T item) {
        this.item = item;
    }
}
public static void main(String[] args) {
        // 黄金的盒子
        Box<Gold> boxGold=new Box<Gold>();
        boxGold.setItem(new Gold());
        Gold gold=boxGold.getItem();
        System.out.println(gold);
        gold.lignt();
        System.out.println("----------------------------------");
        // 硬币的盒子
        Box<Coin> coinGold=new Box<Coin>();
        coinGold.setItem(new Coin());
        Coin coin=coinGold.getItem();
        System.out.println(coin);
        coin.buy();
}
结果:
黄金
金闪闪...
---------------------------------
硬币
拿着一块买两包辣条...

总结：泛型可以提高代码的复用性

```

* 能够掌握泛型方法

  * 定义格式：

    * 修饰符 <代表泛型的变量> 返回值类型 方法名(参数){ }

    * 特点：调用方法时，确定泛型的类型

```
/**
 * @author xiaozhao
 * 泛型方法
 */
public class Container {

    public <E> void add(E element) {
        System.out.println(element.getClass());
    }
}
public static void main(String[] args) {
        Container container=new Container();
        container.add("123");
        container.add(1);
        container.add(1.8);
    }
结果:
class java.lang.String
class java.lang.Integer
class java.lang.Double

```

* 能够掌握含有接口的泛型

  * 定义格式：

    * 修饰符 interface接口名<代表泛型的变量> { }

```
* 定义类时确定泛型的类型
public interface Collection<E> {
    void add(E e);
    E get();
}

public class CollectionImpl implements Collection<String>{
    private String e;
    @Override
```

```
    public void add(String e) {
        this.e=e;
    }

    @Override
    public String get() {
        return this.e;
    }
}
public static void main(String[] args) {
        Collection<String> c=new CollectionImpl();
        c.add("xiaohei");
        System.out.println(c.get());
    }
结果:
xiaohei

* 始终不确定泛型的类型，直到创建对象时，确定泛型的类型
public interface Collection<E> {
    void add(E e);
    E get();
}
public class CollectionImpl2<E> implements Collection<E>{
    private E e;
    @Override
    public void add(E e) {
        this.e=e;
    }

    @Override
    public E get() {
        return this.e;
    }
}
public static void main(String[] args) {
        Collection<String> c=new CollectionImpl2<String>();
        c.add("xiaohei");
        System.out.println(c.get());

        Collection<Integer> c1=new CollectionImpl2<Integer>();
```

```
        c1.add(2);
        System.out.println(c1.get());
    }
 结果:
xiaohei
2
```

* 能够理解泛型通配符

    * 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

    * 注意：泛型是没有继承关系的

        * Collection<Object> list = new ArrayList<String>(); 这种写法是错的

```
* 演示泛型是没有继承关系的
public class Parent {
}
public class Son extends Parent{
}
public class BoxUtils {
    public static void setBox(Box<Parent> box) {
        System.out.println(box);
    }
}
public static void main(String[] args) {
        Box<Parent> box=new Box<Parent>();
        BoxUtils.setBox(box);
        Box<Son> box1=new Box<Son>();
//      BoxUtils.setBox(box1);// 编译时候出错
    }

* 改为通配符
public class BoxUtils {
    public static void setBox(Box<?> box) {
        System.out.println(box);
    }
}
```

```
public static void main(String[] args) {
        Box<Parent> box=new Box<Parent>();
        BoxUtils.setBox(box);
        Box<Son> box1=new Box<Son>();
        BoxUtils.setBox(box1);// 编译没出错
    }
```

* 能够理解泛型上下限

  * JAVA的泛型中可以指定一个泛型的上限和下限

   * 泛型的上限：

     * 格式： 类型名称 <? extends 类 > 对象名称

     * 意义： 只能接收该类型及其子类

   * 泛型的下限：

     * 格式： 类型名称 <? super类 > 对象名称

     * 意义： 只能接收该类型及其父类

```
public static void main(String[] args) {
        // Object String Integer Number (Number是Integer父类)
        java.util.Collection<Object> c1=new ArrayList<Object>();
        java.util.Collection<String> c2=new ArrayList<String>();
        java.util.Collection<Integer> c3=new ArrayList<Integer>();
        java.util.Collection<Number> c4=new ArrayList<Number>();

        // 上限:只能接收该类型及其子类
//      getCollExtends(c1);// 出错 Object
//      getCollExtends(c2);// 出错String
        getCollExtends(c3);// Integer
        getCollExtends(c4);// Number

```

```
        // 下限:只能接收该类型及其父类
        getCollSuper(c1);//Object
//      getCollSuper(c2);// 出错 String
//      getCollSuper(c3);// 出错Integer
        getCollSuper(c4);//Number
    }

    public static void getCollExtends(java.util.Collection<? extends Number> co

    }

    public static void getCollSuper(java.util.Collection<? super Number> coll)

    }
```

* 统计一个数组中大于某个特定元素的个数

```
* Integer
    * Utils
    public static int statisticsGreaterCount(Integer[] items,Integer item) {
        int count=0;
        for(int var:items) {
            if(var>item) {
                count++;
            }
        }
        return count;
    }
    public static void main(String[] args) {
        Integer[] arr= {1,68,9,8,120};
        int count = Utils.statisticsGreaterCount(arr, 10);
        System.out.println(count);
    }
 结果:
 2
* Gold
```

```java
public class Gold{
    private double weight;

    public Gold() {
        super();
    }

    public Gold(double weight) {
        super();
        this.weight = weight;
    }

    @Override
    public String toString() {
        return "黄金";
    }

    public void lignt() {
        System.out.println("金闪闪...");
    }

    public double getWeight() {
        return weight;
    }

    public void setWeight(double weight) {
        this.weight = weight;
    }

}
 Utils:
    public static int statisticsGreaterCount(Gold[] items,Gold item) {
        int count=0;
        for(Gold var:items) {
            if(var.getWeight()>item.getWeight()) {
                count++;
            }
        }
        return count;
    }
```

```
 public static void main(String[] args) {
        Gold[] golds=new Gold[5];
        for (int i = 0; i < golds.length; i++) {
            golds[i]=new Gold(i+10);
        }
        Gold gold=new Gold(11);
        int count = Utils.statisticsGreaterCount(golds, gold);
        System.out.println(count);

    }
结果:
3

* Coin

public class Coin {
    private int value;

    public Coin() {
        super();
    }

    public Coin(int value) {
        super();
        this.value = value;
    }

    @Override
    public String toString() {
        return "硬币";
    }

    public void buy() {
        System.out.println("拿着一块买两包辣条...");
    }

    public int getValue() {
        return value;
    }

```

```
    public void setValue(int value) {
        this.value = value;
    }
}
public static int statisticsGreaterCount(Coin[] items,Coin item) {
        int count=0;
        for(Coin var:items) {
            if(var.getValue()>item.getValue()) {
                count++;
            }
        }
        return count;
    }
public static void main(String[] args) {
        Coin[] Coins=new Coin[5];
        for (int i = 0; i < Coins.length; i++) {
            Coins[i]=new Coin(i+10);
        }
        Coin Coin=new Coin(12);
        int count = Utils.statisticsGreaterCount(Coins, Coin);
        System.out.println(count);

    }
 结果：
 2

 * 思考：可不可以写个公共的方法复用代码（利用泛型）
 public class Utils2 {
    // 思考：可不可以写个公共的方法（利用泛型）
    public static <T extends Comparable<T>> int statisticsGreaterCount(T[] item
        int count=0;
        for(T var:items) {//边界符号
            if(var.compareTo(item)>0) {// short, int, double, long, float, byte
                count++;
            }
        }
        return count;
    }
}
Gold
```

```
 * 实现Comparable接口
    public class Gold implements Comparable<Gold>
 * 实现compareTo方法
 @Override
    public int compareTo(Gold o) {
        return (int) (weight-o.weight);
    }
Coin:
   * 实现Comparable接口
    public class Coin implements Comparable<Gold>
 * 实现compareTo方法
 @Override
    public int compareTo(Coin o) {
        return value-o.value;
    }
* 测试：工具类Utils2 --> 把Utils改成Utils2测试结果
 结果：效果一样

```

* 能够使用集合完成综合案例
  * 斗地主案例
    * 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

```
* 思路
* 准备一副牌
* 洗牌
* 发牌
  * 创建三个玩家（每个玩家17张牌，每人依次摸1张）
  * 创建底牌集合（最后三张牌）
* 查看牌
```

```
* 叫地主
* 实现
* 实现
/**
 * 牌
 */
public class Card {
    /**
     * 花色
     */
    private String color;
    /**
     * 数字
     */
    private String number;


    public Card() {
        super();
    }
    public Card(String color, String number) {
        super();
        this.number = number;
        this.color = color;
    }
    public String getNumber() {
        return number;
    }
    public void setNumber(String number) {
        this.number = number;
    }
    public String getColor() {
        return color;
    }
    public void setColor(String color) {
        this.color = color;
    }
    @Override
    public int hashCode() {
        final int prime = 31;
```

```
        int result = 1;
        result = prime * result + ((color == null) ? 0 : color.hashCode());
        result = prime * result + ((number == null) ? 0 : number.hashCode());
        return result;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Card other = (Card) obj;
        if (color == null) {
            if (other.color != null)
                return false;
        } else if (!color.equals(other.color))
            return false;
        if (number == null) {
            if (other.number != null)
                return false;
        } else if (!number.equals(other.number))
            return false;
        return true;
    }
    @Override
    public String toString() {
        return (color!=null)?color+number:number;
    }

}

/**
 *
 * 玩家
 */
public class Player {
    private String name;
    public List<Card> cards=new ArrayList<Card>();
```

```java
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}


/**
 *
 * 生成牌盒的工具类
 */
public class CardUtils {
    public static final String[] COLORS= {"♠","♥","♣","♦"};
    public static final String[] NUMBERS= {"2","3","4","5","6","7","8","9","10"
    /**
     * 获取一副牌
     * @return
     */
    public static List<Card> getCards(){
        List<Card> cards=new ArrayList<Card>();
        for (int i = 0; i < COLORS.length; i++) { // 4
            for (int j = 0; j < NUMBERS.length; j++) { //13
                Card card=new Card(COLORS[i],NUMBERS[j]);
                cards.add(card);
            }
        }
        // 大小王
        Card sCard=new Card();
        sCard.setNumber("小王");
        Card bCard=new Card();
        bCard.setNumber("大王");
        cards.add(sCard);
        cards.add(bCard);
        return cards;
    }
}

public class Client {
```

```java
    public static void main(String[] args) {
        // 1 准备一副牌
        List<Card> cards = CardUtils.getCards();
        System.out.println("新牌：");
        System.out.println(cards);

        // 2 洗牌
        Collections.shuffle(cards);
        System.out.println("洗完牌：");
        System.out.println(cards);

        // 3 发牌
        // 创建3个玩家
        Player player1=new Player();
        player1.setName("刘备");
        Player player2=new Player();
        player2.setName("张飞");
        Player player3=new Player();
        player3.setName("关羽");
        // 创建底牌盒子
        List<Card> finalCards=new ArrayList<Card>();
        for (int i = 0; i < cards.size(); i++) {
            Card card=cards.get(i);
            if(i<51) {
                if(i%3==0) {
                    player1.cards.add(card);
                }else if(i%3==1) {
                    player2.cards.add(card);
                }else if(i%3==2) {
                    player3.cards.add(card);
                }
            }else { // 51,52,53
                // 最后三张给底牌
                finalCards.add(card);
            }
        }
        // 查看牌
        System.out.println("查看牌：");
        System.out.println(player1.getName()+":"+player1.cards);
        System.out.println(player2.getName()+":"+player2.cards);
```

```
        System.out.println(player2.getName()+":"+player3.cards);
        System.out.println("底牌"+":"+finalCards);

        // 叫地主
        System.out.println("恭喜"+player1.getName()+"获得抢到地主");
        player1.cards.addAll(finalCards);
        finalCards.clear();
        System.out.println("查看牌: ");
        System.out.println(player1.getName()+":"+player1.cards);
        System.out.println(player2.getName()+":"+player2.cards);
        System.out.println(player2.getName()+":"+player3.cards);
        System.out.println("底牌"+":"+finalCards);

    }
}

结果:
新牌:
[♠2, ♠3, ♠4, ♠5, ♠6, ♠7, ♠8, ♠9, ♠10, ♠J, ♠Q, ♠K, ♠A, ♥2, ♥3, ♥4, ♥5, ♥6, ♥7, ♥
洗完牌:
[♦5, ♣4, ♣K, ♣6, ♠K, ♠5, ♦6, ♦J, ♦K, ♣J, ♠J, ♥7, ♠3, 小王, ♥5, ♣7, ♠10, 大王, ♥
查看牌:
刘备:[♦5, ♣6, ♦6, ♣J, ♠3, ♣7, ♥J, ♣A, ♠6, ♠4, ♥A, ♠7, ♦10, ♦4, ♣3, ♥4, ♠Q]
张飞:[♣4, ♠K, ♦J, ♠J, 小王, ♠10, ♥3, ♥2, ♥K, ♦Q, ♥Q, ♠8, ♠9, ♣9, ♥10, ♥6, ♦7]
张飞:[♣K, ♠5, ♦K, ♥7, ♥5, 大王, ♠2, ♣10, ♣2, ♦2, ♦8, ♣5, ♥9, ♦3, ♥8, ♣8, ♦9]
底牌:[♦A, ♠A, ♣Q]
恭喜刘备获得抢到地主
查看牌:
刘备:[♦5, ♣6, ♦6, ♣J, ♠3, ♣7, ♥J, ♣A, ♠6, ♠4, ♥A, ♠7, ♦10, ♦4, ♣3, ♥4, ♠Q, ♦A,
张飞:[♣4, ♠K, ♦J, ♠J, 小王, ♠10, ♥3, ♥2, ♥K, ♦Q, ♥Q, ♠8, ♠9, ♣9, ♥10, ♥6, ♦7]
张飞:[♣K, ♠5, ♦K, ♥7, ♥5, 大王, ♠2, ♣10, ♣2, ♦2, ♦8, ♣5, ♥9, ♦3, ♥8, ♣8, ♦9]
底牌:[]
```