1

2

```
// 求一个整数各位数的和
public static int addAllDigits(int target){
    int res = 0;        // result int
    String target2 = "";
    target2 = target2 + target;
    char[] targetcopy = target2.toCharArray();
    for (int i = 0; i<targetcopy.length; i++){
        res+= ((int)targetcopy[i] - (int)'0');
    }
    return res;
}
```

3

1. 编写程序，从控制台或对话框任意输入一个英文字符串，统计字符串中每个英文字母出现的次数并输出到控制台（大小写不敏感）。

```
public static void main(String[] args) {
    // 字符串输入
    String str;
    str = new Scanner(System.in).next();

    // 字符串处理
    // 1. 变成小写
    // 2. 转换为char数组进行排序
    str.toLowerCase();
    char[] str2 = str.toCharArray();
    Arrays.sort(str2);

    // 处理输出
    int count = 1;
    int i = 1;
    for (; i < str2.length; i++) {
        if (i == (str2.length - 1)) {
            if (str2[i] == str2[i - 1]) {
                System.out.println(str2[i - 1] + " " + ++count);
            } else {
                System.out.println(str2[i-1] + " " + count);
                System.out.println(str2[i] + " " + 1);
            }
        } else {
            if (str2[i] == str2[i - 1]) {
                count++;
            } else {
                System.out.println(str2[i - 1] + " " + count);
                count = 1;
            }
```

```
        }

    }
}
```

2. 假设一个车牌号码由三个大写字母和后面的四个数字组成。编写一个程序. 随机生成5个不重复的车牌号码。

```
public static void main(String[] args) {
    String[] res = new String[]{"","","","","",};
    for (int i = 0; i<5;i++){
        StringBuffer str = generateLicense();
        res[i] = str.toString();
        if (i > 0){
            for (int j = i-1; j >=0; j--){
                if (res[j].toString().equals(res[i].toString())){
                    i--;
                    break;
                }
            }
        }
    }

    for (int i = 0; i< 5;i++){
        System.out.println(res[i]);
    }

}



public static StringBuffer generateLicense(){
    char c1 =(char)(int)(Math.random()*26+65);
    char c2 =(char)(int)(Math.random()*26+65);
    char c3 =(char)(int)(Math.random()*26+65);
    char c4 = (char)(Math.random()*10+'0');
    char c5 = (char)(Math.random()*10+'0');
    char c6 = (char)(Math.random()*10+'0');
    char c7 = (char)(Math.random()*10+'0');

    StringBuffer str = new StringBuffer();

 str.append(c1).append(c2).append(c3).append(c4).append(c5).append(c6).append(c7
);
    return str;
}
```

4

实现下面二个方法，并在Test3里添加入口main函数测试运行。

Tips：注意检查输入参数row的值，当输入负数，0时如何处理也考虑进来，如何处理这种情况不做要求，可以简单地打印出提示信息，或者抛出异常。但最简单的办法就是当出现这些边界条件，直接返回null引用就行了。由这个方法的调用者去处理。另外也不考虑当row的值太大导致内存溢出的情况。

```
/**
 * 创建一个不规则二维数组
 * 第一行row列
 * 第二行row - 1列
 * ...
 * 最后一行1列
 * 数组元素值都为默认值
 *
 * @param row 行数
 * @return 创建好的不规则数组
 */
public static int[][] createArray(int row) {
    if (row <= 0) {
        System.out.println("row error");
        return null;
    }
    int[][] arr = new int[row][];
    int length = row;
    for (int i = 0; i < length; i++, row--) {
        arr[i] = new int[row];
    }
    return arr;
}

/**
 * 逐行打印出二维数组，数组元素之间以空格分开
 *
 * @param a
 */
public static void printArray(int[][] a) {
    for (int i = 0; i < a.length; i++) {
        for (int j = 0; j < a[i].length; j++) {
            System.out.print(a[i][j]);
        }
        System.out.println("");
    }

}
```

5

```
public static void main(String[] args) {
        ArrayList list = new ArrayList();
        // 增加：add() 将指定对象存储到容器中
        list.add("计算机网络");
        list.add("现代操作系统");
```

```
        list.add("java编程思想");
        list.add("java核心技术");
        list.add("java语言程序设计");
        System.out.println(list);
        Iterator it = list.iterator();
        while (it.hasNext()) {
            String next = (String) it.next();
            System.out.println(next);
        }
}


public static void main(String[] args) {
    ArrayList list = new ArrayList();
    // 增加：add() 将指定对象存储到容器中
    list.add("计算机网络");
    list.add("现代操作系统");
    list.add("java编程思想");
    list.add("java核心技术");
    list.add("java语言程序设计");
    System.out.println(list);

    for (Iterator it = list.iterator(); it.hasNext();) {
        //迭代器的next方法返回值类型是Object，所以要记得类型转换。
        String next = (String) it.next();
        System.out.println(next);
    }
}
```

6

```
package homework.ch19.p1;

/**
 * 迭代器接口，用于遍历组件树里的每一个组件. 注意这不是java.util.Iterator接口
 */
interface Iterator<T> {
    /**
     *  是否还有元素
     * @return 如果元素还没有迭代完，返回true;否则返回false
     */
    boolean hasNext();

    /**
     * 获取下一个元素
     * @return  下一个元素
     */
    Object next();
}

/**
 * 数组迭代器
 */
class  ArrayIterator<T> implements Iterator<T>{
    private int pos = 0;
```

```
    private T[] a = null;

    public ArrayIterator(T[] array){
        a = array;
    }

    @Override
    public boolean hasNext() {
        return !(pos >= a.length);
    }

    @Override
    public T next() {
        if(hasNext()){
            T c = a[pos];
            pos ++;
            return c;
        }
        else
            return null;
    }
}

/**
 * 容器类，内部用Object[]保存元素
 */
class Container<T> implements Iterator<T>{
    private T[] elements;
    private int elementsCount = 0;
    private int size = 0;

    public Container(int size){
        elements = (T[])new Object[size];
        this.size = size;
    }

    public boolean add(T e){
        if(elementsCount < size){
            elements[elementsCount ++] = e;
            return true;
        }
        else{
            return  false;
        }
    }

    /**
     * 返回容器的迭代器
     * @return
     */
    public Iterator<T> iterator(){
        return new ArrayIterator<>(elements);
    }

    @Override
    public boolean hasNext() {
        return false;
    }
```

```
    @Override
    public Object next() {
        return null;
    }
}

public class Test{
    public static void main(String[] args){
        Container<String> container = new Container<>(6);
        container.add("12");
        container.add("34");
        container.add("56");
        container.add("78");
        container.add("9");
        Iterator it = container.iterator();
        while (it.hasNext()){
            String s = (String)it.next();
            if( s != null)
                System.out.println(s);
        }
    }
}
```

7

```
public class TwoTuple<T extends Comparable<T>,K extends Comparable<K>>
implements Comparable<TwoTuple<T,K>>{
    private T first;
    private K second;

    public TwoTuple(T first, K second) {
        this.first = first;
        this.second = second;
    }

    @Override
    public boolean equals(Object obj) {
        TwoTuple<T,K> o = (TwoTuple<T,K>) obj;
        return Objects.equals(first, o.first) &&
Objects.equals(second,o.second);
    }

    @Override
    public String toString() {
        return "TwoTuple{" +
                "first=" + first +
                ", second=" + second +
                '}';
    }

    @Override
    public int compareTo(TwoTuple<T,K> o) {
        int cmp = this.getFirst().compareTo(o.getFirst());
        if (cmp==0){
```

```
            return this.getSecond().compareTo(o.getSecond());
        }
        return cmp;
    }

    public T getFirst() {
        return first;
    }

    public void setFirst(T first) {
        this.first = first;
    }

    public K getSecond() {
        return second;
    }

    public void setSecond(K second) {
        this.second = second;
    }
}



public class test {
    public static void main(String[] args) {
        TwoTuple<Integer,String> twoTuple1 =new TwoTuple<>(1, "ccc");
        TwoTuple<Integer,String> twoTuple2 =new TwoTuple<>(1, "bbb");
        TwoTuple<Integer,String> twoTuple3 =new TwoTuple<>(1, "aaa");
        TwoTuple<Integer,String> twoTuple4 =new TwoTuple<>(2, "ccc");
        TwoTuple<Integer,String> twoTuple5 =new TwoTuple<>(2, "bbb");
        TwoTuple<Integer,String> twoTuple6 =new TwoTuple<>(2, "aaa");
        List<TwoTuple<Integer,String>> list = new ArrayList<>();
        list.add(twoTuple1);
        list.add(twoTuple2);
        list.add(twoTuple3);
        list.add(twoTuple4);
        list.add(twoTuple5);
        list.add(twoTuple6);
        //测试equals，contains方法是基于equals方法结果来判断
        TwoTuple<Integer,String> twoTuple10 =new TwoTuple<>(1, "ccc"); //内容
=twoTuple1
        System.out.println(twoTuple1.equals(twoTuple10)); //应该为true
        if(!list.contains(twoTuple10)){
            list.add(twoTuple10);  //这时不应该重复加入
        }
        //sort方法是根据元素的compareTo方法结果进行排序，课测试compareTo方法是否实现正确
        Collections.sort(list);
        for (TwoTuple<Integer, String> t: list) {
            System.out.println(t);
        }
        TwoTuple<TwoTuple<Integer,String >,TwoTuple<Integer,String >> tt1 =
                new TwoTuple<>(new TwoTuple<>(1,"aaa"),new TwoTuple<>(1,"bbb"));
        TwoTuple<TwoTuple<Integer,String >,TwoTuple<Integer,String >> tt2 =
                new TwoTuple<>(new TwoTuple<>(1,"aaa"),new TwoTuple<>(2,"bbb"));
        System.out.println(tt1.compareTo(tt2)); //输出-1
        System.out.println(tt1);
    }
```

```
}
```

8

```
// 下面程序是一个泛型容器Container<T>的定义，
// 该容器是对ArrayList的一个封装，实现了四个公有的方法add、remove、size、get。
// 请实现泛型Container<T>的一个线程安全的版本SynchronizedContainer<T>，
// 只需要实现与Container<T>一样的4个公有方法。要求必须用synchronized同步语句块或Lock锁实现
class SynchronizedContainer<T>{
    private List<T> elements = new ArrayList<>();
    private final Lock lock = new ReentrantLock();
    /**
     * 添加元素
     *
     * @param e 要添加的元素
     */
    public void add(T e) {
        lock.lock();
        elements.add(e);
        lock.unlock();
    }

    /**
     * 删除指定下标的元素
     *
     * @param index 指定元素下标
     * @return 被删除的元素
     */
    public T remove(int index) {
        lock.lock();
        T ele = elements.remove(index);
        lock.unlock();
        return ele;
    }

    /**
     * 获取容器里元素的个数
     *
     * @return 元素个数
     */
    public int size() {
        return elements.size();
    }

    /**
     * 获取指定下标的元素
     *
     * @param index 指定下标
     * @return 指定下标的元素
     */
    public T get(int index) {
        return elements.get(index);
    }
}
```

9

实现一个线程安全的同步队列SyncQueue，模拟多线程环境下的生产者消费者机制，SyncQueue的定义如下：

```
/**
 * 一个线程安全同步队列，模拟多线程环境下的生产者消费者机制
 * 一个生产者线程通过produce方法向队列里产生元素
 * 一个消费者线程通过consume方法从队列里消费元素
 * @param <T> 元素类型
 */
```

要求实现基于二个版本的SyncQueue，第1个版本为类名为SyncQueue1，第1个版本为类名为SyncQueue2，其功能要求分别为：

1）SyncQueue1实现生产者线程和消费者线程**轮流**生产数据和消费数据，即如果队列不为空，则生产者线程必须等到消费者线程将队列里的数据消费完后才能向队列生产数据；如果队列为空，则消费者线程必须等待生产者线程向队列生产数据后才能消费数据。

2）SyncQueue2要求有些区别：即**生产者线程不管队列是否为空，随时可以向队列生产数据**；消费者线程则在队列为空时，必须等待生产者线程向队列生产数据后才能消费数据。

(1)

```
public class SyncQueue1<T> {
    /**
     * 保存队列元素
     */
    private ArrayList<T> list = new ArrayList<>();


    /**
     * 生产数据
     *
     * @param elements 生产出的元素列表，需要将该列表元素放入队列
     * @throws InterruptedException
     */
    public synchronized void produce(List<T> elements) throws
InterruptedException {
        if (!list.isEmpty()) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        list.addAll(elements);
```

```
        System.out.println("Produce: " + list.toString());
        this.notifyAll();

    }

    /**
     * 消费数据
     *
     * @return 从队列中取出的数据
     * @throws InterruptedException
     */
    public synchronized List<T> consume() {
        if (list.isEmpty()) {
            try {
                this.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        List<T> consumeList = new ArrayList<T>(list);
        list.clear();
        System.out.println("Consume: " + consumeList.toString());
        this.notifyAll();
        return consumeList;
    }


    public static void main(String[] args) {
        SyncQueue1<Integer> syncQueue = new SyncQueue1<>();
        Runnable produceTask = () -> {
            int count = 10;
            while (count-- != 0) {
                try {
                    List<Integer> list = new ArrayList<>();
                    int elementsCount = (int) (Math.random() * 10) + 1;
                    for (int i = 0; i < elementsCount; i++) {
                        int r = (int) (Math.random() * 10) + 1;
                        list.add(r);
                    }
                    syncQueue.produce(list);
                    Thread.sleep((int) (Math.random() * 5) + 1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        Runnable consumeTask = () -> {
            int count = 10;

            while (count-- != 0) {
                try {
                    List<Integer> list = syncQueue.consume();
                    Thread.sleep((int) (Math.random() * 10) + 1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
```

```
        };

        ExecutorService es = Executors.newFixedThreadPool(2);
        es.execute(produceTask);
        es.execute(consumeTask);
        es.shutdown();
        while (!es.isTerminated()) {
        }
    }
}
```

(2)

```
public class SyncQueue2<T> {
    /**
     * 保存队列元素
     */
    private ArrayList<T> list = new ArrayList<>();


    /**
     * 生产数据
     *
     * @param elements 生产出的元素列表，需要将该列表元素放入队列
     * @throws InterruptedException
     */
    public void produce(List<T> elements) {
        synchronized (this){
            list.addAll(elements);
            this.notifyAll();
            System.out.println("Produce: " + list.toString());
        }

    }

    /**
     * 消费数据
     *
     * @return 从队列中取出的数据
     * @throws InterruptedException
     */
    public List<T> consume() {
        synchronized (this){
            if (list.isEmpty()) {
                try {
                    this.wait();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            List<T> consumeList = new ArrayList<T>(list);
            list.clear();
            System.out.println("Consume: " + consumeList.toString());
            return consumeList;
```

```
        }
    }


    public static void main(String[] args) {
        SyncQueue2<Integer> syncQueue = new homework.ch30.SyncQueue2<>();
        Runnable produceTask = () -> {
            while (true) {
                try {
                    List<Integer> list = new ArrayList<>();
                    int elementsCount = (int) (Math.random() * 10) + 1;
                    for (int i = 0; i < elementsCount; i++) {
                        int r = (int) (Math.random() * 10) + 1;
                        list.add(r);
                    }
                    syncQueue.produce(list);
                    Thread.sleep((int) (Math.random() * 5) + 1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        Runnable consumeTask = () -> {
            while (true) {
                try {
                    List<Integer> list = syncQueue.consume();
                    Thread.sleep((int) (Math.random() * 10) + 1);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };

        ExecutorService es = Executors.newFixedThreadPool(2);
        es.execute(produceTask);
        es.execute(consumeTask);
        es.shutdown();
        while (!es.isTerminated()) {
        }
    }


}
```