面向葫芦娃编程：

一、主要类说明：

- 【主类】class GourdSort：

```
// 主类
public class GourdSort{
    // 静态变量
    static int babynum;
    static GourdBaby[] brothers;
    static Cards cards;
    // 静态块
    static {..}
    // 主方法
    public static void main(String[] args)..
    // 初始化方法
    public static void initialize()..
    // orchestration 排序方法
    public static void orchestrationSort()..
    // choreography 排序方法
    public static void choreographySort()..
    // 结果显示方法
    public static void printResult()..
}
```

成员简介：

数据成员包括【葫芦娃的个数】babynum，【葫芦娃类数组】brothers，以及【卡牌数组】cards；

方法成员包括【主方法】main，【初始化方法】initialize，【Orchestration排序算法】orchestrationSort，【choreography排序算法】choreographySort，以及【排序结果打印方法】printResult；

部分方法功能简介：

main：

```
public static void main(String[] args)
{
    System.out.println("-----The first way to sort as orchestration-----");
    initialize();
    orchestrationSort();
    printResult();

    System.out.println("-----The second way to sort as choreography-----");
    initialize();
    choreographySort();
    printResult();
}
```

=> 初始化葫芦娃数组，并依次调用两种形式的排序算法对数组进行排序，最后打印排序结果

initialize：

```
public static void initialize()
{
    cards.shuffle();
    for(int i=0;i<babynum;i++)
    {
        brothers[i] = new GourdBaby(cards.deal());
        brothers[i].setpos(i);
    }
    System.out.println("The original order is as below:");
    for(int i=0;i<babynum;i++)
    {
        brothers[i].numberOff();
    }
}
```

=> 调用Cards类的成员cards的【洗牌方法】shuffle随机打乱次序，再依次通过【发牌方法】deal将初始的战队序号分给葫芦娃

orchestrationSort：=> 详见“二、Orchestration排序”

choreographySort：=> 详见“三、Choreography排序”

- 【葫芦娃类】class GourdBaby：

```
//葫芦娃类
class GourdBaby extends Thread // 继承线程类
{
    // 私有成员变量
    private Color color;  // 颜色
    private int pos;  // 队列中的位置
    private static GourdBaby[] brothers;  // 葫芦娃队列
    // 构造方法
    GourdBaby(int v)
    // 线程执行方法
    public void run()
    // 其他方法
    public void setpos(int p)
    public int getRank()
    public void setRank(int v)
    // 报数方法
    public void numberOff()
    // 比较方法
    public boolean cmp(GourdBaby other)
    // 交换方法
    public void swap(GourdBaby other)
    // 线程同步方法
    private synchronized void lookAround()
    private synchronized boolean doneAll()
    // 排队方法（排序算法调用）
    public static void queue(int idx,GourdBaby[] bros)
    // 多线程排序
    private static void threadSort()
    // 冒泡排序
    private static void bubbleSort()
    //其他排序
    private static void quickSort()
    private static void mergeSort()
    // 内部类
    class Color
}
```

成员简介：

数据成员包括【葫芦娃颜色属性】color，【葫芦娃位序】pos，以及【静态数组】brothers；

方法成员主要包括【线程执行方法】run，【排行相关方法】setpos,getRank,setRank，【报数方法】numberoff，【比较排行方法】cmp，【交换次序方法】swap，【观察左右调整次序方法】lookAround，【判断是否排队完成方法】doneAll，【排序算法调用方法】queue，【一系列排序算法】threadSort,bubbleSort,quickSort等；

部分方法功能简介：

numberoff：

```
public void numberOff()
{
    System.out.println("I am the "+color.getColor()+" Baby!");
}
```

=> 我采用的是输出“I am the Color Baby!”来报数，例如大娃会输出“I am the Red Baby!”，二娃会输出“I am the Orange Baby!”，以此类推；

cmp 和 swap:

```
public boolean cmp(GourdBaby other)
{
    if(this.getRank()<other.getRank())
        return true;
    return false;
}
public void swap(GourdBaby other)
{
    int rk1 = this.getRank();
    int rk2 = other.getRank();
    this.setRank(rk2);
    other.setRank(rk1);
}
```

=> cmp会比较【自己】this和【兄弟】other的排行，若自己排行更小则返回true，反之返回false；

=> swap则交换【自己】this和【兄弟】other的排行，以达到间接交换次序的目的；

run,lookAround,queue等：=> 详见“三、chereography排序”

- 【爷爷类】class GrandFather：

```
//爷爷类
class GrandFather implements GourdCmpSwap // 实现了葫芦娃比较/交换接口
{
    private GourdBaby[] brothers;
    // 比较方法
    public boolean cmp(GourdBaby gb1,GourdBaby gb2)
    // 交换方法
    public void swap(GourdBaby gb1,GourdBaby gb2)
    // 指挥方法（排序算法调用）
    public void command(int idx,GourdBaby[] bros)
    // 冒泡排序
    private void bubbleSort()
    // 其它排序
    private void quickSort()
    private void mergeSort()
}
```

成员简介：

数据成员包括【静态葫芦娃数组】brothers；

方法成员主要包括【比较两葫芦娃排行方法】cmp，【交换两葫芦娃次序方法】swap，【指挥方法/排序算法调用方法】command，【一系列排序算法】bubbleSort,quickSort等；

部分方法功能简介：

cmp 和 swap: （实现了【葫芦娃比较/交换接口】GourdCmpSwap中的抽象方法）

```
public boolean cmp(GourdBaby gb1,GourdBaby gb2)
{
    if(gb1.cmp(gb2))
        return true;
    return false;
}
public void swap(GourdBaby gb1,GourdBaby gb2)
{
    gb1.swap(gb2);
}
```

=> cmp会比较【两个葫芦娃】gb1和gb2的排行，若前者排行更小则返回true，反之返回false；

=> swap则交换【两个葫芦娃】gb1和gb2的排行，以达到间接交换次序的目的；

command：=> 详见“二、Orchestration排序”

- 【卡牌类】class Cards：

```
//卡牌类
class Cards
{
    int num = 0;
    int next = 0;
    boolean[] visited;
    int[] cards;
    // 构造方法
    Cards(int n)
    // 洗牌方法
    public void shuffle()
    // 发牌方法
    public int deal()
    // 清局方法
    public void clear()
}
```

成员简介：

=> 数据成员包括【牌数】num，【当前发牌序号】next，【访问互斥数组】visited，以及【卡牌数组】cards

=> 方法成员包括【洗牌方法】shuffle，【发牌方法】deal，【重发方法】clear

部分方法简介：

=> 略

二、Orchestration排序：

- 【算法思想】

让【爷爷】GrandFather类对象gf，通过【比较两个葫芦娃的排行】cmp，和【交换两个葫芦娃的次序】swap两个基本操作，【指挥葫芦娃】command，【调用一系列的基于比较的排序算法】bubbleSort,quickSort等，对【葫芦娃数组】brothers进行排序

- 【具体实现】

orchestrationSort：

```
public static void orchestrationSort()
{
    int choice = 0;

    GrandFather gf = new GrandFather();
    gf.command(choice,brothers);
}
```

=> 在【主类】GourdSort中，通过调用【orchestration排序算法】orchestrationSort，【选择排序算法的序号】choice，然后调用【爷爷类的指挥方法】gf.command，就可以通过choice选择的排序算法对【葫芦娃数组】brothers进行排序了

=> **排序算法只需要通过更改choice的值即可替换**

command：

```
public void command(int idx,GourdBaby[] bros)
{
    brothers = bros;
    switch(idx)
    {
    case 0: bubbleSort(); break;
    case 1: quickSort(); break;
    case 2: mergeSort(); break;
    default: bubbleSort(); break;
    }

}
```

=> 赋值【葫芦娃数组的引用】brothers

=> 通过switch语句，调用相应的排序算法，此处默认为冒泡算法

bubbleSort：

```
private void bubbleSort()
{
    int n = brothers.length;
    for(int i=0;i<n-1;i++)
        for(int j=0;j<n-i-1;j++)
        {
            if(!cmp(brothers[j],brothers[j+1]))
                swap(brothers[j],brothers[j+1]);
        }
}
```

=> 爷爷通过cmp和swap两个基本操作，利用冒泡的思想，对葫芦娃数组完成了排序

=> 此处仅写了冒泡算法为例，其他的排序算法暂未实现（但只要是基于比较的排序算法，通过cmp和swap两个操作一定都可以实现）

三、Choreography排序：

- 【算法思想】

让【葫芦娃】brothers[i]通过【比较操作】cmp，前后比较【相邻葫芦娃】brother[i-1]或brother[i+1]的排行，相应地进行【交换】swap，即可协作以完成排序，

但此处不同于Orchestration排序的是，除了【调用一系列基于比较的排序算法】bubbleSort,quickSort之外，由于葫芦娃的动作的“相似性”以及“可并发性”，可以利用【多线程编程】的技术，实现一个基于比较的【多线程排序算法】threadSort，这样可能使得“协作效率”大大提高；

- 【具体实现】

choreographySort：

```
public static void choreographySort()
{
    int choice = 0;

    GourdBaby.queue(choice, brothers);
    try{Thread.sleep(500);}
    catch(InterruptedException e) {System.out.println(e.toString());}
}
```

=> 在【主类】GourdSort中，通过调用【choreography排序算法】choreographySort，【选择排序算法的序号】choice，然后调用【葫芦娃类的静态排队方法】GourdbBaby.queue，就可以通过choice选择的排序算法对【葫芦娃数组】brothers进行排序了

=> 排序算法只需要通过更改choice的值即可替换

queue：

```
public static void queue(int idx,GourdBaby[] bros)
{
    brothers = bros;
    switch(idx)
    {
    case 0: threadSort(); break;
    case 1: bubbleSort(); break;
    case 2: quickSort(); break;
    case 3: mergeSort(); break;
    default: threadSort(); break;
    }
}
```

=> 赋值【葫芦娃数组的引用】brothers

=> 通过switch语句，调用相应的排序算法，此处默认为多线程算法

threadSort和bubbleSort：

```
private static void threadSort()
{
    for(int i=0;i<brothers.length;i++)
    {
        brothers[i].start();
    }
}
private static void bubbleSort()
{
    int n = brothers.length;
    for(int i=0;i<n-1;i++)
        for(int j=0;j<n-i-1;j++)
        {
            if(!brothers[j].cmp(brothers[j+1]))
                brothers[j].swap(brothers[j+1]);
        }
}
```

=> threadSort直接通过调用每个【葫芦娃的线程启动函数】start，即可开启线程

=> bubbleSort原理同“二、Orchestration排序”，不再赘述

run：

```
public void run()
{
    while(true)
    {
        lookAround();
        if(doneAll())
            break;
    }
}
```

=> 线程执行函数，每个葫芦娃不断【观察周围】lookAround，并相应地调整次序

=> 直到【判断所有的人都站好了】即可退出线程

lookAroud和doneAll：

```
private synchronized void lookAround()
{
    if(pos>0)
    {
        if(cmp(brothers[pos-1]))
            swap(brothers[pos-1]);
    }
    if(pos<brothers.length-1)
    {
        if(!cmp(brothers[pos+1]))
            swap(brothers[pos+1]);
    }
}
private synchronized boolean doneAll()
{
    boolean doneall = true;
    for(int i=0;i<brothers.length;i++)
    {
        if(brothers[i].getRank()!=i)
        {
            doneall = false;
            break;
        }
    }
    return doneall;
}
```

=> lookAround即观察相邻左右兄弟的次序，并相应地进行交换调整（当然最左边和最右边的只能观察一边）

=> 若干次迭代后，如果【所有葫芦娃都站好队了】doneAll，则运行当前线程退出；

四、Java特性/语言机制使用：

- 【继承】

  如：【葫芦娃类】GourdBaby继承了【线程类】Thread，如下图所示：

```
// 葫芦娃类
class GourdBaby extends Thread // 继承线程类
{
```

- 【封装】

  如：【葫芦娃类】GourdBaby封装了内部类【颜色类】Color，如下图所示：

```
    // 内部类
    class Color
    {
        private String col = null;
        private int val = 7;
        Color(int v)
        public String getColor()
        public int getVal()
        public void setVal(int v)
    }
}
```

- 【多态】

  如：【线程类】Thread的【线程执行方法】run()被子类【葫芦娃类】GourdBaby重写/覆盖了，如下图所示：

```
// 线程执行方法
public void run()
{
    while(true)
    {
        lookAround();
        if(doneAll())
            break;
    }
}
```

- 【接口】

  如：【爷爷类】GrandFather实现了【葫芦娃比较/交换接口】GourdCmpSwap，如下图所示：

```
// 爷爷类
class GrandFather implements GourdCmpSwap // 实现了葫芦娃比较/交换接口
{
    private GourdBaby[] brothers;
    // 比较方法
    public boolean cmp(GourdBaby gb1,GourdBaby gb2)
    // 交换方法
    public void swap(GourdBaby gb1,GourdBaby gb2)
```

- 【构造器】

  如：【葫芦娃类】Gourdbaby的构造器，用于初始化位置和颜色，如下图所示：

```
// 构造方法
GourdBaby(int v)
{
    pos = -1;
    color = new Color(v);
}
```

- 【静态变量】

  如：【主类】GourdSort的静态成员变量，便于【静态主函数】main中使用，如下图所示：

```
// 主类
public class GourdSort{
    // 静态变量
    static int babynum;
    static GourdBaby[] brothers;
    static Cards cards;
```

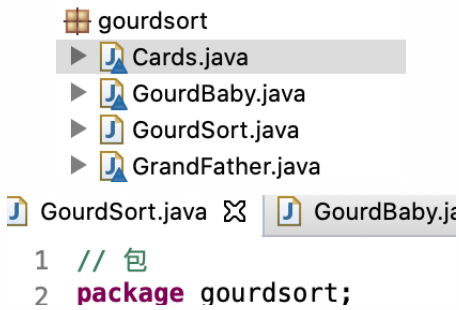
- 【静态块】

  如：【主类】GourdSort的静态块，用于对静态成员变量进行初始化，如下图所示：

```
// 静态块
static {
    babynum = 7;
    brothers = new GourdBaby[babynum];
    cards = new Cards(babynum);
}
```

- 【包】

  我将每个类都单独存于一个java源文件中，将它们放到同一个【包】gourdsort下，并在每个源文件的第一条语句对包进行导入，如下图所示：



- 【修饰符】

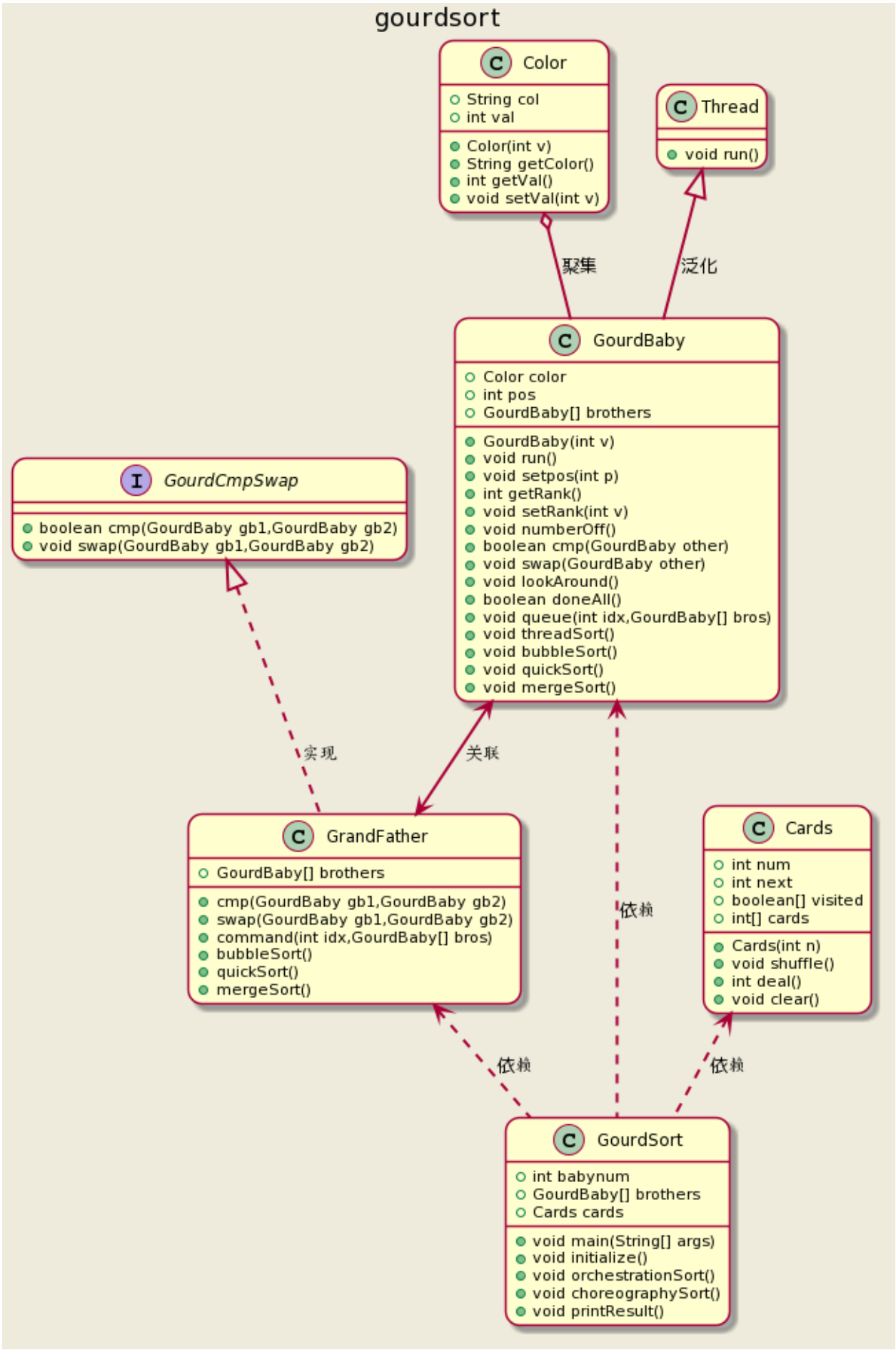
如：【主类】GourdSort中的各访问控制符，静态方法修饰符，以及【葫芦娃比较/交换接口】GourdCmpSwap中的抽象方法修饰符，如下图所示：

```
    // 结果显示方法
    public static void printResult()
}

// 葫芦娃比较/交换操作接口
interface GourdCmpSwap{
    public abstract boolean cmp(GourdBaby gb1,GourdBaby gb2);
    public abstract void swap(GourdBaby gb1,GourdBaby gb2);
}
```

五、类图展示：

- 【PNG】


gourdsort
Color
String col
int val
Color(int v)
String getColor()
int getVal()
void setVal(int v)
Thread
void run()
聚集
泛化
GourdBaby
Color color
int pos
GourdBaby[] brothers
GourdBaby(int v)
void run()
void setpos(int p)
int getRank()
void setRank(int v)
void numberOff()
boolean cmp(GourdBaby other)
void swap(GourdBaby other)
void lookAround()
boolean doneAll()
void queue(int idx,GourdBaby[] bros)
void threadSort()
void bubbleSort()
void quickSort()
void mergeSort()
GourdCmpSwap
boolean cmp(GourdBaby gb1,GourdBaby gb2)
void swap(GourdBaby gb1,GourdBaby gb2)
实现
关联
GrandFather
GourdBaby[] brothers
cmp(GourdBaby gb1,GourdBaby gb2)
swap(GourdBaby gb1,GourdBaby gb2)
command(int idx,GourdBaby[] bros)
bubbleSort()
quickSort()
mergeSort()
Cards
int num
int next
boolean[] visited
int[] cards
Cards(int n)
void shuffle()
int deal()
void clear()
依赖
依赖
依赖
GourdSort
int babynum
GourdBaby[] brothers
Cards cards
void main(String[] args)
void initialize()
void orchestrationSort()
void choreographySort()
void printResult()

- 【URL】

  http://www.plantuml.com/plantuml/png/XLH1JzH05BxtLqpYWIMme1m9CS2A7arOuiLucBPlsybEP-hhT05N3npmO8XU38aI5JKUlAW9CIQA-cTOTZtv5tpjG9aIS2yDyxutt_lc-nwxc1cE9a-aPuIHm5YaSmmpZSQRoBf2fHnvmb8UT7a4RRD9BGY1cPrWWoVxW-qFezylHcztZdwy7-vl_JtSETx_DZXyDlp-UVZcu-3dkubPxup49v88TQx2bfOQsPMLbPNb6osdemG3ZGgGpLrrW0pkvw02M4BKwskn2BeAiepDkHfZhi9r5CQ0OWPpyBn0ScgwLJoehT7qlIcX3CdmDrMUU5Cbi4odk_UOZzh4W9atrUBqV1RGbmuzBKAMSA4QROD2HTJ9CSecJn2XX15SYWVGE2rf36B8378Zj2gcLaWGQmGT8K_ZpHgG4hDPXIoNHUsHezlA8ztMiA3usaUaEhli0LMrKJJsJgTWhgg96HWYAF5qqX96O5QvwZejzbpZ8DjymDkTJj7cQos1AnOaQQEQoKeL5SawlnIJMdUNoi1Tib0hM9Aoaa6nvr2A4E76zBa7LpcO6867DLFztFSbk2NYehsfLHB0oFOulfTUa-SsyS9qwr2FovCWAbDiI-6ThPzvUOTBrqewsalEdBLITD-9nm7BBRGBMxX-yWTicCegig0dCc4WB44wsfsr2rnEKvM2EEzq99pg2y5H5n0PrlOEkGflyYAh_cKxLiyyygzDkwSvugRenxK4EaberF_IhQTO2_1STaG2sE41i9AcbQHjKj1trtIS7clLMDqqp1t15XxEpBYcpGy-lHu-_UBPV0bkDbb5CNzyi3VOsVNiFxIkGQEjbt_s7djdunQ8kFhrcZ_wzMhqTTTp9btSuDnlDg_NbMqVZBQUU-M6N7nv4LIO9_8V

- 【UML】

```
@startuml
title  gourdsort
'skinparam packageStyle rect/' 加入这行代码，样式纯矩形'/
skinparam backgroundColor #EEEBDC
skinparam roundcorner 20
skinparam sequenceArrowThickness 2
'skinparam handwritten true

class GourdSort{
+int babynum
+GourdBaby[] brothers
+Cards cards
+void main(String[] args)
+void initialize()
+void orchestrationSort()
+void choreographySort()
+void printResult()
}

class GourdBaby{
+Color color
+int pos
+GourdBaby[] brothers
+GourdBaby(int v)
+void run()
+void setpos(int p)
+int getRank()
+void setRank(int v)
+void numberOff()
+boolean cmp(GourdBaby other)
+void swap(GourdBaby other)
+void lookAround()
+boolean doneAll()
```

```
+void queue(int idx,GourdBaby[] bros)
+void threadSort()
+void bubbleSort()
+void quickSort()
+void mergeSort()
}

class Color{
+String col
+int val
+Color(int v)
+String getColor()
+int getVal()
+void setVal(int v)
}

class Thread{
+void run()
}

class Cards{
+int num
+int next
+boolean[] visited
+int[] cards
+Cards(int n)
+void shuffle()
+int deal()
+void clear()
}

class GrandFather{
+GourdBaby[] brothers
+cmp(GourdBaby gb1,GourdBaby gb2)
+swap(GourdBaby gb1,GourdBaby gb2)
+command(int idx,GourdBaby[] bros)
+bubbleSort()
+quickSort()
+mergeSort()
}

interface GourdCmpSwap{
+boolean cmp(GourdBaby gb1,GourdBaby gb2)
+void swap(GourdBaby gb1,GourdBaby gb2)
}

GourdCmpSwap <|.. GrandFather:实现
Thread <|-- GourdBaby:泛化
Color o-- GourdBaby:聚集
GourdBaby <.. GourdSort:依赖
GrandFather <.. GourdSort:依赖
GourdBaby <--> GrandFather:关联
```

```
Cards <.. GourdSort:依赖
@enduml
```

六、运行示例及说明：

- 【终端运行示例】

```
-----The first way to sort as orchestration-----
The original order is as below:
I am the Purple Baby!
I am the Orange Baby!
I am the Green Baby!
I am the Yellow Baby!
I am the Cyan Baby!
I am the Red Baby!
I am the Blue Baby!
The result order is as below:
I am the Red Baby!
I am the Orange Baby!
I am the Yellow Baby!
I am the Green Baby!
I am the Cyan Baby!
I am the Blue Baby!
I am the Purple Baby!
-----The second way to sort as choreography-----
The original order is as below:
I am the Yellow Baby!
I am the Orange Baby!
I am the Purple Baby!
I am the Blue Baby!
I am the Cyan Baby!
I am the Red Baby!
I am the Green Baby!
The result order is as below:
I am the Red Baby!
I am the Orange Baby!
I am the Yellow Baby!
I am the Green Baby!
I am the Cyan Baby!
I am the Blue Baby!
I am the Purple Baby!
```

- 【运行说明】

  => 两种不同形式的排序过程依次打印

  => 排序过程首先打印【初始序列报数】original order，再打印【排好序报数】result order

  => 另外，值得说明的是，本人在测试过程中，发现偶尔【多线程算法】会打印出错，经过调试后发现是由于主线程的【结果打印函数】**printResult**先于若干葫芦娃子线程执行导致的，即使本人在主线程中添加了【线程睡眠】**Thread.sleep(500)**，也只是降低了发生打印错误的概率，仍然有可能发生打印错误

  => 因此，希望老师或助教在测试代码时，如果遇到打印错误，请再测试**1**次或**n**次，勿轻易判定为代码不合格，谢谢！