

第3次Java实验

姓名：李哲彦

学号：37220222203675

日期：2024年5月13日

院系：信息学院

专业：计算机科学与技术

实验一：实现自己的字符串类

解题思路

声明一个对象 *mystring* 用来存储字符串内容。由于我们不知道存储的字符串长度为多少，故初始化时先将其设为 0。之后，再需要修改字符串长度时，可以将其直接复制为一个新的相应长度的字符串。

核心代码

首先，声明对象，并设置构造函数。默认构造函数则直接设置为长度为0 的空串。

```
char []mystring;
MyString(){
    mystring = new char[0];
}
MyString(char[] ch){
    mystring = ch;
}
```

由于我们要查找某个字符第一次出现的下标，故使用 *for* 循环从前往后找，若找到该字符则立即返回下标。如果找到末尾也没找到，说明该字符不存在，返回 -1 表示不存在。

```
public int indexof(char ch){

    for(int i = 0; i < mystring.length; i++){
        if(mystring[i] == ch){
            return i;
        }
    }
    return -1;
}
```

若要根据下表返回对应位置的字符，直接访问字符数组返回即可。但是此处要注意非法情况，即下标为负或超过字符串长度的情况，此时返回 `#`。

```
public char charAt(int x){
    if(x ≥ mystring.length || x < 0) return (char) '#';
    else return mystring[x];
}
```

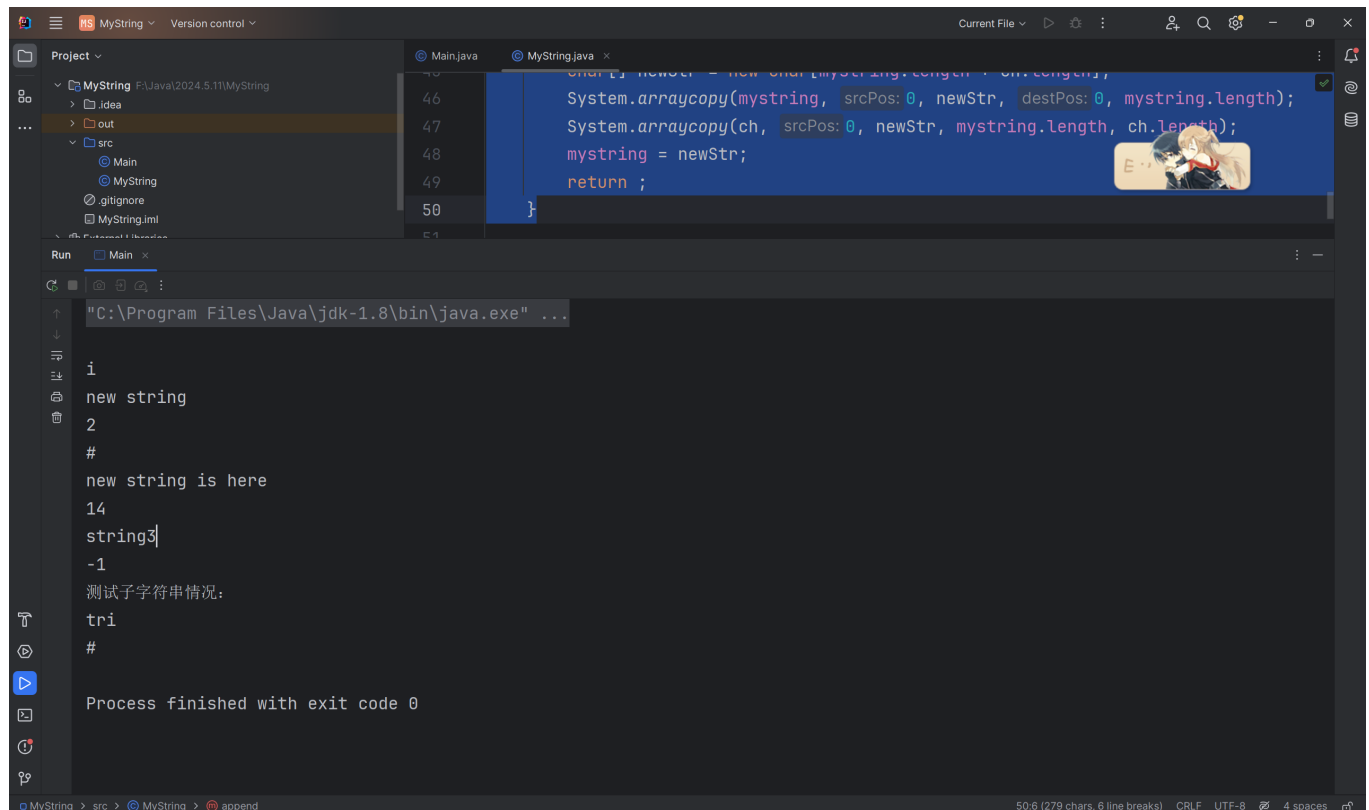
对于获取子串的操作，我们可以临时创建一个 `char` 数组，长度为对应的目标子串长度。当然，我们也可以直接创建一个新的 `mystring` 类，并使用 `append` 操作。但由于这种写法会需要不断重新创建、复制字符串，效率较低。此操作也需要注意判断下标非法时的情况，返回一个内容为 `#` 的字符串表示错误。

```
public MyString substring(int begin, int end){
    if(begin < 0 || end ≥ mystring.length){ //非法情况
        return new MyString(new char[] { '#'});
    }
    char[] str = new char[end - begin];
    for(int i = max(begin, 0); i < min(mystring.length, end); i++){
        str[i - begin] = mystring[i];
    }
    return new MyString(str);
}
```

最后对于增加字符的操作，由于需要增长字符串长度，故此处选择新建一个长度为最新值的字符串，然后进行复制与拼接操作。

```
public void append(char[] ch){
    char[] newStr = new char[mystring.length + ch.length];
    System.arraycopy(mystring, 0, newStr, 0, mystring.length);
    System.arraycopy(ch, 0, newStr, mystring.length, ch.length);
    mystring = newStr;
    return ;
}
```

最后得到如下运行结果：



The screenshot shows an IDE with a project named 'MyString'. The 'src' directory contains 'Main.java' and 'MyString.java'. The 'Run' window shows the execution of 'Main.java'. The output is as follows:

```
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...  
i  
new string  
2  
#  
new string is here  
14  
string3  
-1  
测试子字符串情况:  
tri  
#  
  
Process finished with exit code 0
```

调试过程与心得

本实验，我遇到的最多的问题是边界问题的考虑。例如，在使用下标查找字符的函数 *charAt* 中，我一开始只想到了下标大于字符串长度的情况，导致测试数据无法通过。经过检查才发现我少考虑了下标小于 0 的情况。

然后是获取子字符串的 *substring* 操作。一开始我也没有考虑提供的头尾下标不合法的情况，导致运行异常。

可见，考虑异常状况是多么的重要。今后在编写代码的时候，我一定要多加考虑特殊情况和用户的非法输入。

实验二 橄榄球队

解题思路

按照题目要求创建父类和各个子类的对象。对于计算排名等重名函数，对其进行重载以加以区分。对于球队类，创建一个橄榄球队员的数组来存储所有的球员信息。

核心代码

对于比较排名的功能，使用函数计算二者的排名，以方便与不同职业的球员之间的互相比对。

```
public int compareTo(FootballPlayer oppose){  
    if(this.playerRating() > oppose.playerRating()) return 1;
```

```

        else if(this.playerRating() == oppose.playerRating()) return 0;
        else return -1;
    }

```

对于子类, (在此以 *Quarterback* 类为例), 在将其转化为字符串时, 与父类及其他类不同, 因此进行重载。同时为了代码复用, 可以直接使用父类的 *toString* 函数以输出一些基础信息, 然后再加上该类特有的信息。

```

@Override
public String toString() {
    return super.toString() + ", Completion percentage: " +
        completionPercentage() + ", Average Passing Yards Per Game: "
        + averagePassingYardsPerGame() + ", Average Touch Downs Per Game: "
        + averageTouchDownsPerGame() +
        ", Player's Rating: " + playerRating();
}

```

对于其 *getRating* 方法也要进行重载, 以计算其特有的排名计算方法。

```

@Override
public double playerRating() {

    return averageTouchDownsPerGame() +
        (completionPercentage() * 100.0) + (averagePassingYardsPerGame() /
        5.0);
}

```

对于构造函数 (以有参构造函数为例), 需要先调用父类(*super*)的构造函数, 然后再初始化该类的特有信息。

```

public Quarterback(String playerName, String position, String team, int
    gamesPlayed, int passAttempts, int passCompleted, int touchdownsPassing, int
    totalYardsPassing) {
    super(playerName, position, team, gamesPlayed); // 父类的有参构造方法
    this.passAttempts = passAttempts;
    this.passCompleted = passCompleted;
    this.touchdownsPassing = touchdownsPassing;
    this.totalYardsPassing = totalYardsPassing;
}

```

由于其余子类实现内容基本相同, 仅计算公式不同, 不再赘述。

对于球队的 *FoorballTeam* 类, 首先创建各个对象, 下标初始化为 -1 , 表示当前队员为空。

对于添加球员的操作，只需要后移一位下标，然后赋值即可。但是需要考虑球员人数达到上限以及传入的球员是“空球员”（在此以名字为空或指向空内存进行探测）的问题。同时，返回一个字符串。若为空字符串，则表示添加成功，否则返回对应问题。

```
public String addPlayer(FootballPlayer player){
    if(currentPlayerIndex == players.length - 1){
        System.out.println("The number of players is out of the team size!");
        return "The number of players is out of the team size!";
    }
    if(player == null || player.getPlayerName() == ""){
        System.out.println("The player is null! ");
        return "The player is null! ";
    }

    currentPlayerIndex++;
    players[currentPlayerIndex] = player;
    return "";
}
```

对于通过所在位置来查找球员的功能，可以输入其职业，然后通过比对来添加至一个字符串 *s* 中。此处使用 *StringBuilder* 来提高效率。最后返回一个完整的字符串，不同球员的信息之间以换行符分隔。

在测试类中，基于题目中提供的测试内容外，本人也额外追加了一些关于非法问题的测试，最后得到的输出内容如下（不与测试样例的输入完全相同，因为输入各项参数的比例要同时满足的话较难构造）：

```
The number of players is out of the team size!
Information of 1st team:
Team Name: XMU Argos, Team Owner: Mr. Pinto

Team members:
Name: Phil Sims, Position: Quarter Back, Team: Giants, Completion percentage:
0.8, Average Passing Yards Per Game: 50.0, Average Touch Downs Per Game: 3.0,
Player's Rating: 93.0
Name: Jim Brown, Position: Running Back, Team: Browns, Average Yards Per Game:
100.0, Average Yards Per Attempy: 20.0, Average Touch Downs Per Game 3.0,
Player's Rating: 43.0
Name: Spider Lockart, Position: Defensive Back, Team: Giants, Average Tackles
Per Game: 5.0, Average Interceptions Per Game: 2.5, Average Forced Fumbles Per
Game: 1.5, Player's Rating: 78.0

Information of 2nd team:
Team Name: Canton Stompers, Team Owner: Mrs. Pinto
```

Team members:

Name: Steve Young, Position: Quarter Back, Team: 49ers, Completion percentage: 0.5, Average Passing Yards Per Game: 68.0, Average Touch Downs Per Game: 5.0, Player's Rating: 68.6

Name: Saquon Barkley, Position: Running Back, Team: Giants, Average Yards Per Game: 78.0, Average Yards Per Attempt: 19.5, Average Touch Downs Per Game 4.0, Player's Rating: 39.1

Name: Aqib Talib, Position: Defensive Back, Team: Rams, Average Tackles Per Game: 8.0, Average Interceptions Per Game: 5.0, Average Forced Fumbles Per Game: 2.0, Player's Rating: 134.0

Testing comparing two players rating

Comparing Phil Sims with Steve Young

Phil Sims has a higher rating

Testing Finding player by position

Running Back:

Name: Jim Brown, Position: Running Back, Team: Browns, Average Yards Per Game: 100.0, Average Yards Per Attempt: 20.0, Average Touch Downs Per Game 3.0, Player's Rating: 43.0

The position is in a wrong format!

Error detection

The number of players is out of the team size!

The "find player by position" is wrong!

Process finished with exit code 0

调试过程与心得

本次实验最开始时，我对父类与子类的关系还停留在“了解规则”的程度。在编写本次实验之后，我对父类以及子类的关系有了更加清晰的认知。尤其是在编写橄榄球队 *FootballTeam* 这一类的时候，一开始我还在思考如何存储不同职业的队员，遇到了难点。后来才想起来可以直接使用父类来代表所有子类，大大方便了编码。可见父子类在特定场景下的方便之处。

实验三 设计一盘井字棋

解题思路

本程序主要分为三类。首先是父类 *People*，可以存储姓名、年龄等通用信息。当然也可以根据实际需要添加更多的内容，比如胜利场数等。然后是子类 *Chesser*，主要功能是下棋，能够修改棋盘的内容。最后是棋盘，用一个二维数组来存储棋盘上当前的状况，同时方便判断获胜情况。

核心代码

对于 *People* 类，本次主要存储三个信息：姓名、年龄、当前游戏获胜次数。

```
People(String name, String age, int totalWintimes){
    this.name = name;
    this.age = age;
    this.totalWintimes = totalWintimes;
}
```

对于 *Chesser* 类，主要保存当前 *Chesser* 是黑还是白，然后提供一接口落子。如果落子的输入情况非法，则输出错误信息，然后返回 -1 表示错误。

```
public int Walk(int x, int y, Chessbox chessbox){
    String str = chessbox.addChess(x, y, this.Side);
    System.out.println(str);
    if(!Objects.equals(str, "")) return -1;
    else return 1;
}
```

对于 *Cheebox* 类，主要存储棋盘状况，然后提供接口输出棋盘。在接受落子信息时，需要判断是否非法。主要考虑落在棋盘外面（非法坐标）以及同一格已经有其他棋子的情况，然后返回错误信息。

```
public String addChess(int x, int y, String side){ // 该函数只下子，不检查胜负！
    if(x > 3 || y > 3 || x ≤ 0 || y ≤ 0){
        return "Out of size!";
    }
    if(side ≠ "White" && side ≠ "Black"){
        return "The format of side is wrong!";
    }
    if(s[x][y] ≠ 0){
        return "There is already a chess on the position! Please input again!"; // 返回错误信息
    }
    if(Objects.equals(side, "White")) s[x][y] = 1;
    else s[x][y] = -1;
    return "";
}
```

对于检查胜者的情况，此处以行检查为例，只需要检查该方向上的三个格子加起来是否为 -3 或 3 即可。

```
// 检查行方向
for(int i = 1; i ≤ 3; i++){
    sum = 0;
    for(int j = 1; j ≤ 3; j++){
        sum += s[i][j];
    }
    if(sum == 3) return "The white is the winner!";
    else if(sum == -3) return "The black is the winner!";
}
```

最后编写测试类，模拟两个玩家下棋的过程，测试各个类的正确性，得到的输出如下（白色子为 O, 黑色子为 X）:

```
Player information:
Name: Xiaomin, Age: 10, times win: 0
Name: Xiaomei, Age: 20, times win: 0
Game start!
The chessbox now is:

. . .
. . .
. . .

Now is player1's turn! (Black)
Please input your position to walk! (row col)
1 1

The chessbox now is:
X. . .
. . .
. . .

Game continue!
Now is player2's turn! (White)
Please input your position to walk! (row col)
1 1
There is already a chess on the position! Please input again!
4 5
Out of size!
2 2

The chessbox now is:
```


X. . .

.0. .

. . .

Game continue!

Now is player1's turn! (Black)

Please input your position to walk! (row col)

1 2

The chessbox now is:

X.X. .

.0. .

. . .

Game continue!

Now is player2's turn! (White)

Please input your position to walk! (row col)

3 3

The chessbox now is:

X.X. .

.0. .

. .0.

Game continue!

Now is player1's turn! (Black)

Please input your position to walk! (row col)

1 3

The chessbox now is:

X.X.X.

.0. .

. .0.

The black is the winner!

Press 1 to continue, 2 to stop!

2

Game over! The result is:

Player information:

Name: Xiaomin, Age: 10, times win: 1

```
Name: Xiaomei, Age: 20, times win: 0
```

```
Process finished with exit code 0
```

调试过程与心得

本次实验的主要调试过程是特殊情况考虑的不够完全，没有考虑到同一个格子的重复落子情况，导致在实际测试中出现已有棋子被覆盖的问题，这显然是不合法的。其次是实际输出与信息存储方式的转换。最开始我直接使用了二维 *char* 数组来存储棋盘的具体信息。虽然在输出打印的时候比较方便，但是这位判断棋盘胜负状况带来了困难。改用为二维 *int* 数组之后，我们就只需要判断一下求和值是否为 3 即可，且对棋盘的输出打印并没有增加太多的难度，只需要额外判断一下数字即可。

可见，在存储一些信息时，我们不一定要存储其最终输出的形态，而是寻找一个转换过程方便，利于其他方法的实现。

实验四 从多边形到其他形状

解题思路

首先写父类 *Polygon*（这里假设其为凸多边形），然后拆分为多个三角形来计算面积，周长则利用各个点的坐标（按照顺时针或逆时针顺序）计算来得出。对于特殊的形状，则根据其几何性质计算得到所有点的坐标，然后计算各种信息即可。

核心代码

首先写 *Point* 类，主要设置 *setLocation* 与 *getDistance* 两个功能，以方便设置坐标与求解两个点之间的距离。

```
public void setLocation(double x, double y){
    this.x = x;
    this.y = y;
    return ;
}
public double distance(Point b){
    return (double)Math.sqrt((this.x - b.x) * (this.x - b.x) + (this.y - b.y)
    *(this.y - b.y));
}
```

对于不规则的凸多边形类。构造时输入所有顶点的坐标，来创建该多边形。

```
public Polygon(Point[] points){
    this.points = points;
}
```

```
}
```

接着是计算面积。对于一个凸多边形，我们可以选中其中心点（这里设为原点），然后将其拆分为多个三角形。也就是说，我们只需要将每个定点与原点相连，求出 n 条向量，然后将相邻向量叉乘，即得三角形面积。加和起来即得总面积。

```
public double getArea(){
    double area = 0.0;
    int n = points.length;
    for (int i = 0; i < n; i++) {
        int j = (i + 1) % n;
        area += points[i].getX() * points[j].getY() - points[j].getX() *
points[i].getY(); // 相邻向量叉乘，可得三角形面积。
    }
    return Math.abs(area) / 2.0;
}
```

计算周长则较为简单，只需要按照顺时针或逆时针方向依次计算每条边的长度，然后加和即可。

```
public double getPerimeter(){
    double perimeter = 0.0;
    int n = points.length;
    for (int i = 0; i < n; i++) {
        int j = (i + 1) % n;
        perimeter += points[i].distance(points[j]);
    }
    return perimeter;
}
```

对于规则多边形，由于我们在创建对象时只有边数与长度，故需要根据几何性质，根据角度旋转一圈，以得到所有顶点的坐标，固有如下点坐标生成方法：

```
public Point[] GeneratePoint2ds(double sideLength, int sideNum) {
    Point[] points = new Point[sideNum];
    double sita = 2 * Math.PI / (double) sideNum; // 每一个点旋转角度的差值
    double nowArrow = 0.0; // 旋转角度
    double l = sideLength * Math.sin((Math.PI - sita) / 2.0) / Math.sin(sita);
    for (int i = 0; i < points.length; i++) {
        double x = l * Math.sin(nowArrow);
        double y = l * Math.cos(nowArrow);
        nowArrow += sita;
        points[i] = new Point(x, y);
    }
}
```

```
    return points;
}
```

由于在构造对象时，我们必须优先构造 *super* 超类，无法在构造函数中先求点，再构造。因此，我选择了先对父类构造函数传入一空 *Point* 数组，然后再重新修改。

对于五角星类型，我们需要修改计算面积的方法。首先，计算外围五边形的面积，然后可以发现其和五角星的差值是五个完全相同的三角形面积。所以不妨先直接套用正五边形的面积公式，然后再减去五个三角形的面积即可。

```
@Override
public double getArea() {
    double triangleArea = getTriangleArea(sideLength);
    double pentagonArea = getPentagonArea(sideLength);
    return pentagonArea - 5 * triangleArea; // 正五边形减去五个三角形
}

private double getTriangleArea(double sideLength) {
    return 0.5 * sideLength * sideLength * Math.sin(Math.PI * 108 / 180);
}

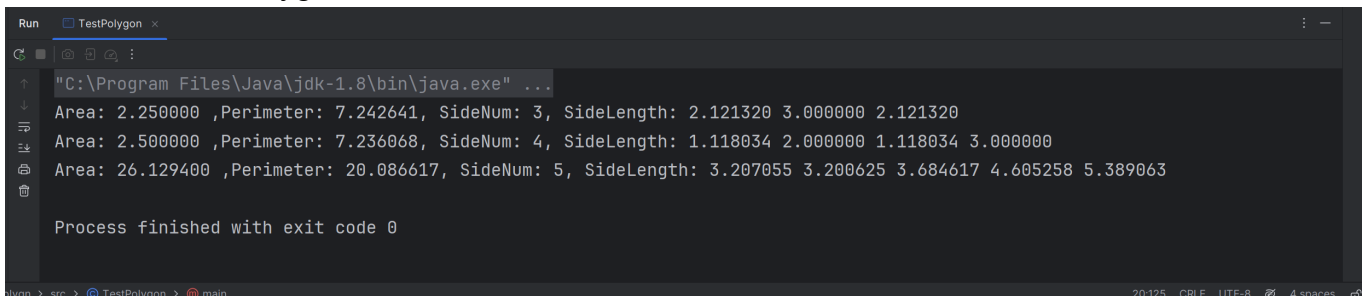
private double getPentagonArea(double sideLength) {
    double sideLength1 = sideLength * Math.cos(Math.PI * 36 / 180) * 2;
    return 5 * Math.tan(Math.PI * 54 / 180) * sideLength1 * sideLength1 / 4; //
    直接用公式
}
```

由于五角星的顶点也在对应的正五边形上，只需要通过五角星边长预处理出正五边形的边长，然后直接使用原正多边形的生成点方法即可。

```
// 利用五角星的几何性质计算
double sideLength1 = sideLength * Math.cos(Math.PI * 36 / 180) * 2;
```

最后编写测试类，对三个类别的分别测试，得到的题目样例输出结果如下：

普通多边形 TestPolygon:



```
Run TestPolygon x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
Area: 2.250000 ,Perimeter: 7.242641, SideNum: 3, SideLength: 2.121320 3.000000 2.121320
Area: 2.500000 ,Perimeter: 7.236068, SideNum: 4, SideLength: 1.118034 2.000000 1.118034 3.000000
Area: 26.129400 ,Perimeter: 20.086617, SideNum: 5, SideLength: 3.207055 3.200625 3.684617 4.605258 5.389063

Process finished with exit code 0
```

规则多边形 TestRegularPolygon:

```
Run TestRegularPolygon x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
Area: 4.273664 ,Perimeter: 9.424778, Side length: 3.141593, SideNum: 3
Area: 110.501591 ,Perimeter: 37.699111, Side length: 3.141593, SideNum: 12
Area: 785395.552745 ,Perimeter: 3141.592600, Side length: 3.141593, SideNum: 1000
Process finished with exit code 0
```

五角星 TestPentagram:

```
Run TestPentagram x
"C:\Program Files\Java\jdk-1.8\bin\java.exe" ...
, Area: 2.126627 ,Perimeter: 5.000000Pentagram with side length: 1.000000, SideNum: 5
Process finished with exit code 0
```

调试过程与心得

本项目的主要调试过程集中在求面积的部分上。在求五角星面积时，我一开始选择了将五角星分成多个小三角形进行求解，但是由于这些小三角形的形状各不相同，需要计算不同的形状带来的后果就是代码长度很长，且不规则形状的计算容易出错。大量小数的计算也会导致结果误差增大。最后，我想到了容斥原理，通过大五边形减去五个小三角形的方法，则只需要计算两种形状即可得到结果。

本次实验总结

通过这次实验，我对父子类的使用变得更加熟悉，理解了二者的关系与一些具体用法。对于代码能力而言，各个项目的调试过程无疑让我意识到了考虑特殊情况的重要性，以及这些情况发生后，对于代码 *debug* 带来的难度与深远影响。因此，以后在编写代码的过程中，我一定要时刻注意各类边界条件、特殊情况，以及用户输入可能的错误形式，尽量在第一次编写后排除大量错误，以减少之后 *debug* 的时间与压力。

其次是父子类的优越之处。这次实验，尤其是在求多边形这一个项目中，父子类对于代码复用的优越性让我印象深刻。比如，求凸多边形面积，对于普通多边形类和正多边形类，我们就可以只需要写一遍求面积公式即可，因为两类多边形的求面积公式是一样的。在以后的编码过程中，我也要好好利用父子类优越的继承机制，减少代码量与失误率。