第六章：类与对象。

第一节：面向对象编程（理解）

面向对象编程的概念
•万物皆对象。
•面向对象指以属性和行为的观点去分析现实生活中的事物。
•面向对象编程指先以面向对象的思想进行分析，然后使用面向对象的编程语言进行表达的过程。
•面向对象编程是软件产业化发展的需求。
•理解面向对象的思想精髓(封装、继承、多态)，至少掌握一种编程语言。

第二节：类和对象及引用==（重中之重）==

类和对象的概念
•对象主要指现实生活中客观存在的实体，在Java语言中对象体现为内存空间中的一块存储区域。
•类简单来就是“分类”，是对具有相同特征和行为的多个对象共性的抽象（所谓的抽象就是，这不是实物可以摸得到的，比如：不会说帮我叫一个外卖，但是外卖是叫不来的，我们要叫的是具体什么吃的喝的，吃的喝的就是外卖的实例化，外卖就是抽象的）描述，在Java语言中体现为一种引用数据类型，里面包含了描述特征/属性的成员变量以及描述行为的成员方法。
•类是用于构建对象的模板，对象的数据结构由定义它的类来决定。

类的定义

•class 类名{
类体;
}
•注意：
通常情况下，当类名由多个单词组成时，要求每个单词首字母都要大写。

这里非常像我们前面一直写的 public class XXX{}

对象的创建

•new 类名();
•注意：
a.当一个类定义完毕后，可以使用new关键字来创建该类的对象，这个
过程叫做类的实例化。
b.创建对象的本质就是在内存空间的堆区申请一块存储区域，用于存放
该对象独有特征信息。

引用的定义

•基本概念
a.使用引用数据类型定义的变量叫做引用型变量，简称为"引用"。
b.引用变量主要用于记录对象在堆区中的内存地址信息，便于下次访问。
•语法格式
类名引用变量名;
引用变量名.成员变量名;

案例题目

•编程实现Person类的定义和使用。

```
/*
    编程实现Person类的定义
 */
public class Person {

  // 数据类型 成员变量名 = 初始值;    - 其中=初始值 通常都省略不写
  String name; // 用于描述姓名的成员变量
  int age;     // 用于描述年龄的成员变量

    // 自定义成员方法实现所有成员变量的打印
    // 返回值类型 方法名称(形参列表) { 方法体; }
  void show() {
    // 成员变量和成员方法都属于类内部的成员，因此可以直接访问成员变量不需要再加引用.的前缀
    System.out.println("我是" + name + ", 今年" + age + "岁了! ");
  }
    // 自定义成员方法实现将姓名修改为参数指定数值的行为
  // String s = "guanyu";
  void setName(String s) {
    name = s;
  }
  // 自定义成员方法实现将年龄修改为参数指定数值的行为
  // int i = 35;
  void setAge(int i) {
    age = i;
  }
    // 自定义成员方法实现将姓名和年龄修改为参数指定数值的行为   下面的方法不推荐使用
  // String s = "liubei";
  // int i = 40;
  void setNameAge(String s, int i) {
    name = s;
    age = i;
  }
    // 自定义成员方法实现可变长参数的使用   看作一维数组使用即可   0 ~ n个
```

```
  void showArgument(int num, String... args) {
    System.out.println("num = " + num);
    for(int i = 0; i < args.length; i++) {
      System.out.println("第" + (i+1) + "个参数为: " + args[i]);
    }
  }
  // 自定义成员方法实现姓名数值的获取并返回的行为
  String getName() {
    return name; // 返回数据并结束当前方法
    // ...   执行不到的
  }
  // 自定义成员方法实现年龄数值的获取并返回的行为
  int getAge() {
    return age;
  }

    public static void main(String[] args) {

    // 1.声明Person类型的引用指向Person类型的对象
    // 数据类型(类名) 引用变量名 = new 类名();
    Person p = new Person();
    // 2.打印对象中的成员变量值
    // 引用变量名.成员变量名
    //System.out.println("我是" + p.name + ", 今年" + p.age + "岁了! "); // null 0
    // 引用变量名.成员方法名(实参列表);
    // 调用方法的本质就是根据方法名跳转过去执行方法体后再跳转回这个位置
    p.show();

    System.out.println("-------------------------------------------------------");
    // 3.修改成员变量的数值
    p.name = "zhangfei";
    p.age = 30;
    // 4.再次打印修改后的数值
    //System.out.println("我是" + p.name + ", 今年" + p.age + "岁了! "); // zhangfei
30
    p.show();

    System.out.println("-------------------------------------------------------");
    // 5.通过成员方法的调用实现成员变量的修改
    p.setName("guanyu");
    p.setAge(35);
    p.show(); // guanyu  35

    System.out.println("-------------------------------------------------------");
    // 6.通过成员方法同时修改姓名和年龄
    //p.setNameAge("liubei", 40);
```

```
    int ia = 40;
    p.setNameAge("liu"+"bei", ia);
    p.show(); // liubei 40

    System.out.println("-------------------------------------------------------");
    // 7.通过成员方法实现可变长参数的打印
    p.showArgument(0);
    System.out.println("-------------------------------------------------------");
    p.showArgument(1, "参数1");
    System.out.println("-------------------------------------------------------");
    p.showArgument(2, "参数1", "参数2");

    System.out.println("-------------------------------------------------------");
    // 8.通过成员方法的调用实现成员变量数值的获取并打印
    String str1 = p.getName();
    System.out.println("获取到的姓名是: " + str1); // liubei
    int ib = p.getAge();
    System.out.println("获取到的年龄是: " + ib); // 40
  }
}
```

成员变量：就是定义在类中的所有变量，类中所有的方法和成员都可以引用。
局部变量：类中的方法自己定义的变量，只有自己可以访问。

案例题目

•编程实现Point类的定义，特征有：横纵坐标(整数)，要求在main方法中声明Point类型的引用指向Point对象并打印特征，然后将横纵坐标修改为3和5后再次打印。

```
/*
    编程实现Point类的定义
 */
public class Point {

  int x; // 用于描述横坐标的成员变量
  int y; // 用于描述纵坐标的成员变量

  // 自定义成员方法实现成员变量数值的打印
  void show() {
    System.out.println("横坐标是: " + x + ", 纵坐标是: " + y);
  }
  // 自定义成员方法实现将横坐标修改为参数指定数值的行为
  // int i = 10;
  void setX(int i) {
```

```
    x = i;
  }
  // 自定义成员方法实现将纵坐标修改为参数指定数值的行为
  // int j = 20;
  void setY(int j) {
    y = j;
  }
  // 自定义成员方法实现int类型的可变长参数使用
  void showArgument(int... args) {
    for(int i = 0; i < args.length; i++) {
      System.out.println("下标为" + i + "的元素是: " + args[i]);
    }
  }
  // 自定义成员方法实现获取横坐标数值并返回的行为
  int getX() {
    return x;
  }
  // 自定义成员方法实现获取纵坐标数值并返回的行为
  int getY() {
    return y;
  }

  public static void main(String[] args) {

    // 1.声明Point类型的引用指向Point类型的对象
    Point p = new Point();
    // 打印成员变量的数值
    //System.out.println("横坐标是: " + p.x + ", 纵坐标是: " + p.y); // 0 0
    p.show();

    System.out.println("-----------------------------------------------");
    // 2.将横纵坐标修改为3和5后再次打印
    p.x = 3;
    p.y = 5;
    //System.out.println("横坐标是: " + p.x + ", 纵坐标是: " + p.y); // 3 5
    p.show();

    System.out.println("-----------------------------------------------");
    // 3.通过调用成员方法实现横纵坐标的修改
    p.setX(10);
    p.setY(20);
    p.show(); // 10 20

    System.out.println("-----------------------------------------------");
    // 4.通过成员方法实现可变长参数的使用
    p.showArgument(1, 2, 3, 4, 5);
```

```
        System.out.println("-----------------------------------------");
        // 5.通过成员方法调用实现横纵坐标的获取
        int ia = p.getX();
        System.out.println("获取到的横坐标是: " + ia); // 10
        int ib = p.getY();
        System.out.println("获取到的纵坐标是: " + ib); // 20
    }
}
```

第三节：成员方法==（重中之重）==

成员方法的定义

•class 类名{
返回值类型成员方法名(形参列表) {
成员方法体;
}
}
•当成员方法名由多个单词组成时，要求从第二个单词起每个单词的首字母大写。

返回值类型的详解

•返回值主要指从方法体内返回到方法体外的数据内容。
•返回值类型主要指返回值的数据类型，可以是基本数据类型，也可以是引用数据类型。
•当返回的数据内容是66时，则返回值类型写int即可
•在方法体中使用return关键字可以返回具体的数据内容并结束当前方法。
•当返回的数据内容是66时，则方法体中写return 66; 即可
•当该方法不需要返回任何数据内容时，则返回值类型写void即可。

形参列表的详解

•形式参数主要用于将方法体外的数据内容带入到方法体内部。
•形式参数列表主要指多个形式参数组成的列表，语法格式如下：
数据类型形参变量名1, 数据类型形参变量名2, ...
•当带入的数据内容是"hello"时，则形参列表写String s 即可
•当带入的数据内容是66和"hello"时，则形参列表写inti, String s 即可
•若该方法不需要带入任何数据内容时，则形参列表位置啥也不写即可。

方法体的详解
•成员方法体主要用于编写描述该方法功能的语句块。
•成员方法可以实现代码的重用，简化代码。

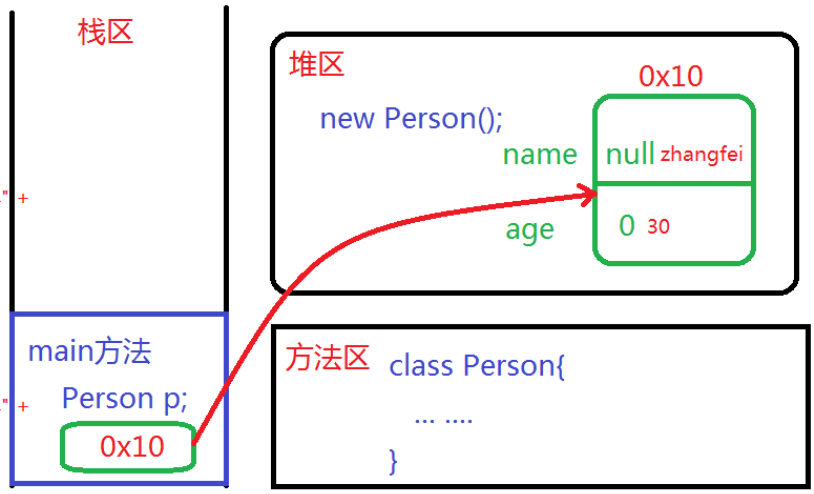
方法的调用
•引用变量名.成员方法名(实参列表);
•实际参数列表主要用于对形式参数列表进行初始化操作，因此参数的个数、类型以及顺序都要完全一致。
•实际参数可以传递直接量、变量、表达式、方法的调用等。
p.show();

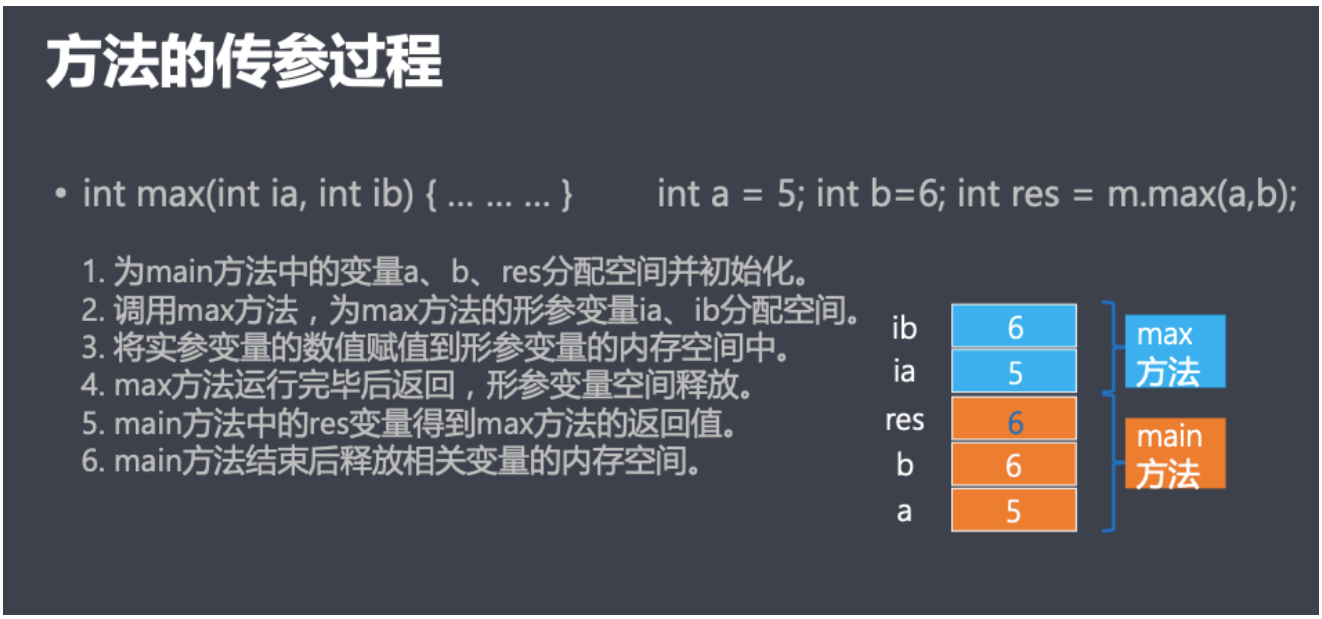
可变长参数
•返回值类型方法名(参数的类型... 参数名)
•方法参数部分指定类型的参数个数是可以改变的，也就是0~n个。
•一个方法的形参列表中最多只能声明一个可变长形参，并且需要放到参数列表的末尾。

```
// 自定义成员方法实现可变长参数的使用   看作一维数组使用即可   0 ~ n个
  void showArgument(int num, String... args) {
    System.out.println("num = " + num);
    for(int i = 0; i < args.length; i++) {
      System.out.println("第" + (i+1) + "个参数为: " + args[i]);
    }
  }
```

```
public class Person{
   String name;
   int age;
   public static void main(String[] args){
      Person p = new Person();
      System.out.println("我是" + p.name + "，今年" +
p.age + "岁了");

      p.name = "zhangfei";
      p.age = 30;
      //打印该对象中的姓名和年龄    zhangfei  30
      System.out.println("我是" + p.name + "，今年" +
p.age + "岁了");
   }
```





参数传递的注意事项

•基本数据类型的变量作为方法的参数传递时，形参变量数值的改变通常不会影响到实参变量的数值，因为两个变量有各自独立的内存空间；

•引用数据类型的变量作为方法的参数传递时，形参变量指向内容的改变会影响到实参变量指向内容的数值，因为两个变量指向同一块内存空间

•当引用数据类型的变量作为方法的参数传递时，若形参变量改变指向后再改变指定的内容，则通常不会影响到实参变量指向内容的改变，因为两个变量指向不同的内存空间。

内存结构之栈区

•栈用于存放程序运行过程当中所有的局部变量。一个运行的Java程序从开始到结束会有多次方法的调用。

•JVM会为每一个方法的调用在栈中分配一个对应的空间，这个空间称为该方法的栈帧。一个栈帧对应一个正在调用中的方法，栈帧中存储了该方法的参数、局部变量等数据。

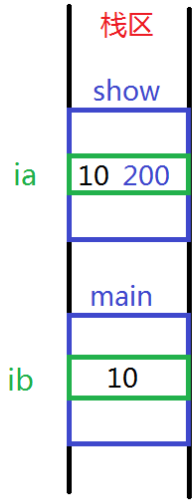
•当某一个方法调用完成后，其对应的栈帧将被清除。

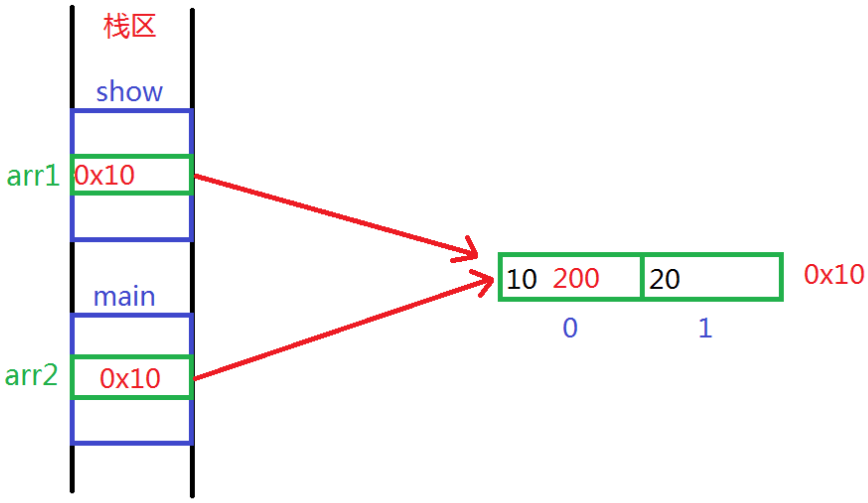
传参的相关概念

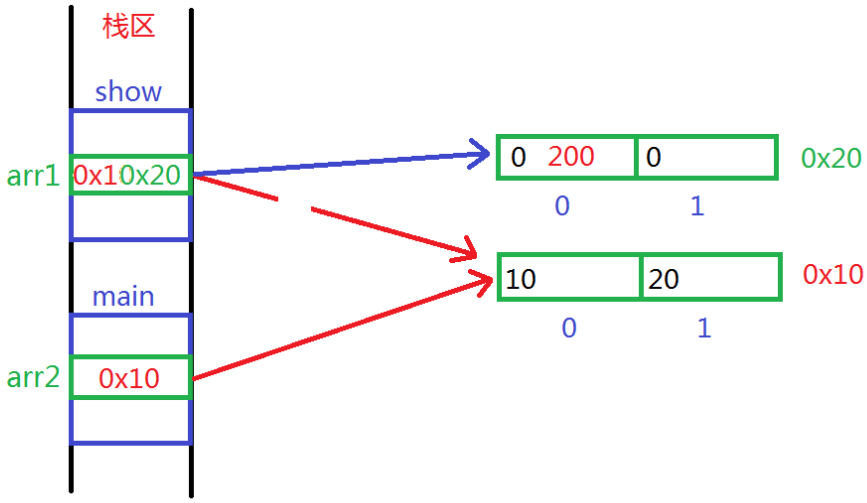
•参数分为形参和实参，定义方法时的参数叫形参，调用方法时传递的参数叫实参。

•调用方法时采用值传递把实参传递给形参，方法内部其实是在使用形参。

•所谓值传递就是当参数是基本类型时，传递参数的值，比如传递i=10，真实传参时，把10赋值给了形参。当参数是对象时，传递的是对象的值，也就是把对象的地址赋值给形参。







第七章：方法和封装

第一节：构造方法（重中之重）

构造方法的基本概念
• class 类名 {
类名(形参列表) {
构造方法体;
}
}
• 构造方法名与类名完全相同并且没有返回值类型，连void都不许有。

```
class Person {
Person() { - Person类中的构造方法
}
}
1234
```

默认构造方法
• 当一个类中没有定义任何构造方法时，编译器会自动添加一个无参空构
造构造方法，叫做默认/缺省构造方法，如：Person(){}
• 若类中出现了构造方法，则编译器不再提供任何形式的构造方法。
构造方法的作用
• 使用new关键字创建对象时会自动调用构造方法实现成员变量初始化工作。

案例题目
• 编程实现Point类的定义并向Point类添加构造方法
Point() 默认创建原点对象
Point(int i, int j) 根据参数创建点对象

```java
/*
    编程实现Point类的定义
 */
public class Point {

  int x; // 用于描述横坐标的成员变量
  int y; // 用于描述纵坐标的成员变量

  // 自定义无参构造方法
  Point() {}
    // 自定义有参构造方法
    Point(int x, int y) {
    this.x = x;
    this.y = y;
  }

  // 自定义成员方法实现特征的打印
  void show() {
    System.out.println("横坐标是: " + x + ", 纵坐标是: " + y);
  }
  // 自定义成员方法实现纵坐标减1的行为
  void up() {
    y--;
  }
  // 自定义成员方法实现纵坐标减去参数指定数值的行为
  void up(int y) {
    this.y -= y;
  }

  public static void main(String[] args) {

    // 1.使用无参方式构造对象并打印特征
    Point p1 = new Point();
    p1.show(); // 0 0

    // 2.使用有参方式构造对象并打印特征
    Point p2 = new Point(3, 5);
    p2.show(); // 3 5

    System.out.println("-------------------------------------");
    // 3.调用重载的成员方法
    p2.up();
    p2.show(); // 3 4
    p2.up(2);
    p2.show(); // 3 2
```

```
    }
}
```

第二节：方法重载（重点）

方法重载的概念
• 若方法名称相同，参数列表不同，这样的方法之间构成重载关系(Overload)。
这里可以联想到我们的println，print等。

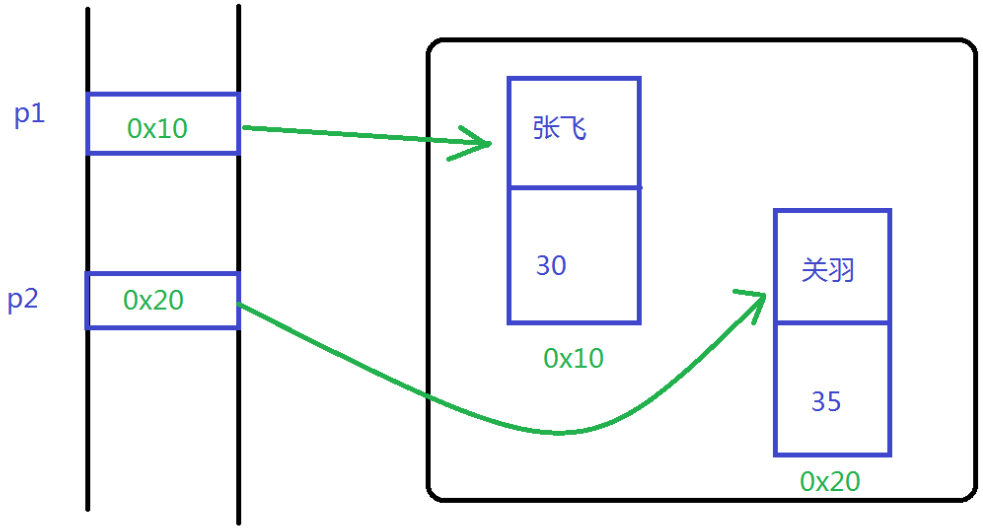
重载的体现形式
• 方法重载的主要形式体现在：参数的个数不同、参数的类型不同、参数的顺序不同，与返回值类型和形参变量名无关，但建议返回值类型最好相同。
• 判断方法能否构成重载的核心：调用方法时能否加以区分。
练习题目
• 编程实现为Point类添加重载的成员方法：
up() – 实现纵坐标减1的功能。
up(int dy) – 实现纵坐标减去参数指定数值的功能。
• 测试重载方法的调用规则
（参考以上代码）

重载的实际意义
• 方法重载的实际意义在于调用者只需要记住一个方法名就可以调用各种不同的版本，来实现各种不同的功能。
• 如：java.io.PrintStream类中的println方法。

第三节： this关键字（原理）

this的基本概念
• 若在构造方法中出现了this关键字，则代表当前正在构造的对象。
• 若在成员方法中出现了this关键字，则代表当前正在调用的对象。
• this关键字本质上就是当前类类型的引用变量。

工作原理
• 在构造方法中和成员方法中访问成员变量时，编译器会加上this.的前缀，而this.相当于汉语中"我的"，当不同的对象调用同一个方法时，由于调用方法的对象不同导致this关键字不同，从而this.方式访问的结果也就随之不同。



使用方式

• 当局部变量名与成员变量名相同时，在方法体中会优先使用局部变量(就近原则)，若希望使用成员变量，则需要在成员变量的前面加上this.的前缀，明确要求该变量是成员变量（重中之重）。

• this关键字除了可以通过this.的方式调用成员变量和成员方法外，还可以作为方法的返回值（重点）。

• 在构造方法的第一行可以使用this()的方式来调用本类中的其它构造方法（了解）。

注意事项

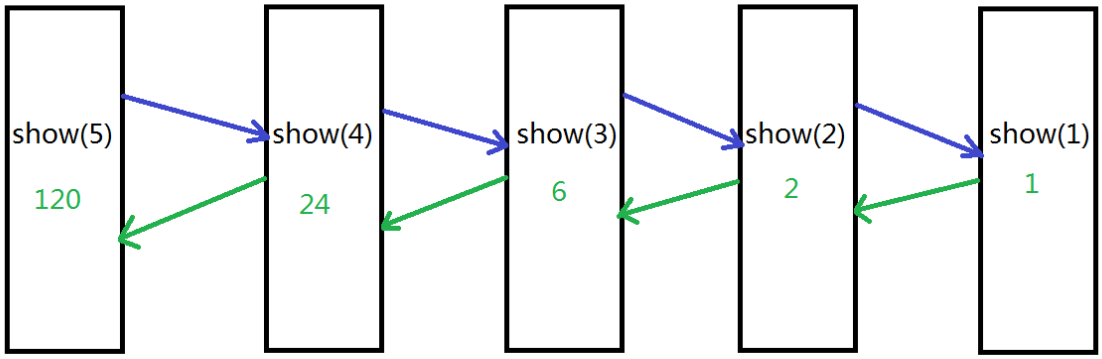
• 引用类型变量用于存放对象的地址，可以给引用类型赋值为null，表示不指向任何对象。

• 当某个引用类型变量为null时无法对对象实施访问（因为它没有指向任何对象）。此时，如果通过引用访问成员变量或调用方法，会产生NullPointerException 异常。

第四节：方法递归调用（难点）

案例题目

• 编程实现参数n的阶乘并返回，所谓阶乘就是从1累乘到n的结果。




```
/*
    编程实现累乘积的计算并打印
 */
public class JieChengTest {

  // 自定义成员方法实现将参数n的阶乘计算出来并返回
  // 1! = 1;      2! = 1*2;    3! = 1*2*3;    ...    n! = 1*2*3*...*n;
  int show(int n) { // int n=5; int n = 4; int n = 3; int n = 2;  int n = 1;
    // 递推的方式
    /*
    int num = 1;
    for(int i = 1; i <= n; i++) {
      num *= i;
    }
    return num;
    */
    /*
        5! = 5 * 4 * 3 * 2 * 1;
      4! = 4 * 3 * 2 * 1;
      3! = 3 * 2 * 1;
      2! = 2 * 1;
      1! = 1;

      5! = 5 * 4!;
      4! = 4 * 3!;
      3! = 3 * 2!;
      2! = 2 * 1!;
      1! = 1;
```

```
      n! = n * (n-1)!;

    */
    // 递归的方式
    // 当n的数值为1时，则阶乘的结果就是1
    /*
    if(1 == n) {
      return 1;
    }
    */
    if(1 == n) return 1;
    // 否则阶乘的结果就是 n*(n-1)!
    return n*show(n-1);
    // show(5) => return 5*show(4); => 120
    // show(4) => return 4*show(3); => 24
    // show(3) => return 3*show(2); => 6
    // show(2) => return 2*show(1); => 2
    // show(1) => return 1;         => 1
  }

  public static void main(String[] args) {

    // 1.声明JieChengTest类型的引用指向该类型的对象
    JieChengTest jct = new JieChengTest();
    // 2.调用方法进行计算并打印
    int res = jct.show(5);
    System.out.println("最终的计算结果是：" + res); // 120
  }
}
```

递归的基本概念

• 递归本质就是指在方法体的内部直接或间接调用当前方法自身的形式。

注意事项

• 使用递归必须有递归的规律以及退出条件。

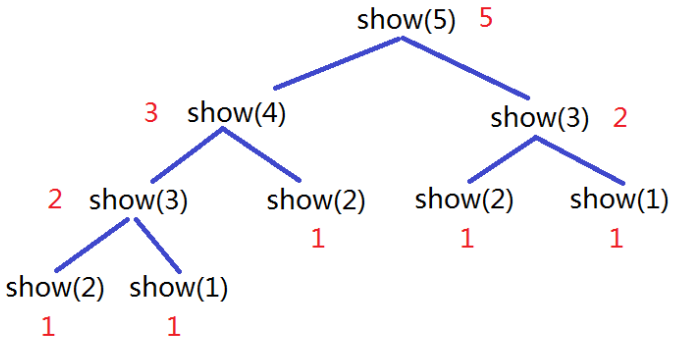
• 使用递归必须使得问题简单化而不是复杂化。

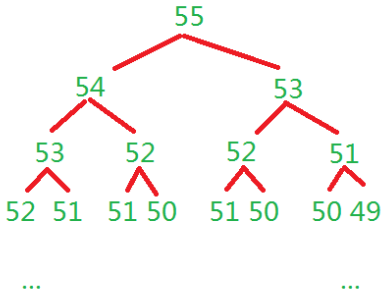
• 若递归影响到程序的执行性能，则使用递推取代之。

案例题目

• 编程实现费式数列中第n项的数值并返回。

• 费式数列： 1 1 2 3 5 8 13 21 .....





1 1 2 3 5 8 13 21 ...

ic

ib

ia

int ia = 1; int ib = 1;

当 n = 1时， return 1; => return ib;

当 n = 2时， return 1; => return ib;

当 n = 3时， return 2; => return ib;

int ic = ia + ib = 2; ia = ib = 1; ib = ic = 2;

当 n = 4时， return 3; => return ib;

int ic = ia + ib = 3; ia = ib = 2; ib = ic = 3;

... ...

综上所述：

当 n = 3时，需要让上述3个变量赋值1轮； 当 n = n 时，需要上述3个变量赋值 n-2 轮

当 n = 4时，需要让上述3个变量赋值2轮；

```
/*
    编程实现费氏数列的计算并打印  功能类/封装类
 */
public class Fee {

  // 自定义成员方法实现费氏数列中第n项数值的计算并返回，n由参数指定
  // 1 1 2 3 5 8 13  21 ....
  int show(int n) { // int n = 5; int n = 4; int n = 3; int n = 2; int n = 1;
    // 1.使用递归的方式进行计算
    /*
    // 当n=1或者n=2时，结果是1
    if(1 == n || 2 == n) {
      return 1;
    }
    // 否则结果是前两项的和
```

```
    return show(n-1) + show(n-2);
    // show(5) => return show(4) + show(3); => 5
    // show(4) => return show(3) + show(2); => 3
    // show(3) => return show(2) + show(1); => 2
    // show(2) => return 1;                 => 1
    // show(1) => return 1;                 => 1
    */
    // 2.使用递推的方式进行计算
    int ia = 1;
    int ib = 1;
    for(int i = 3; i <= n; i++) {
      int ic = ia + ib;
      ia = ib;
      ib = ic;
    }
    return ib;
  }
}
```

第五节：封装（重中之重）get \set

封装的概念

• 通常情况下可以在测试类给成员变量赋值一些合法但不合理的数值，无论是编译阶段还是运行阶段都不会报错或者给出提示，此时与现实生活不符。

• 为了避免上述错误的发生，就需要对成员变量进行密封包装处理，来隐藏成员变量的细节以及保证成员变量数值的合理性，该机制就叫做封装。

封装的实现流程

• 私有化成员变量，使用private关键字修饰。

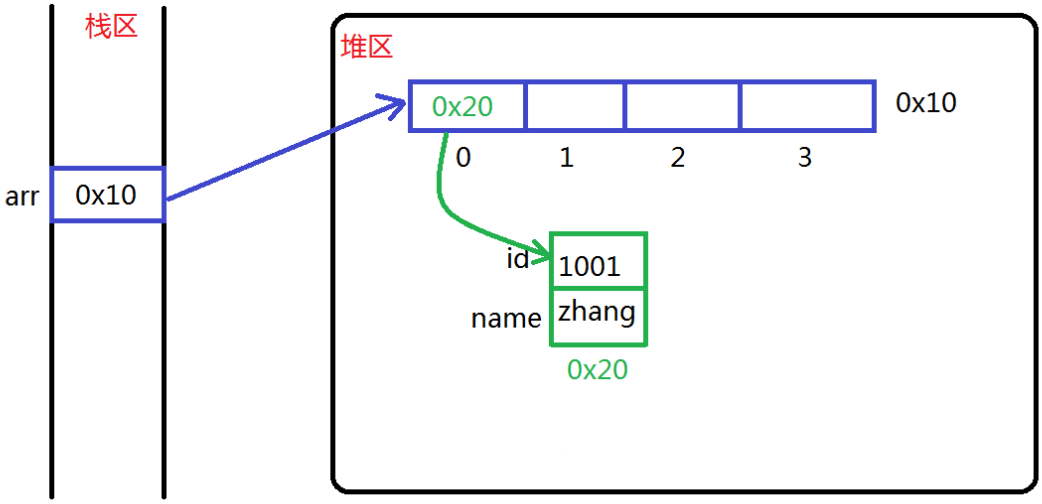
• 提供公有的get和set方法，并在方法体中进行合理值的判断。

• 在构造方法中调用set方法进行合理值的判断。

案例题目

• 提示用户输入班级的学生人数以及每个学生的信息，学生的信息有：学号、姓名，最后分别打印出来。

• 提示：Student[] arr = new Student[num];




```java
/*
    编程实现Student类的封装  封装类
 */
public class Student {

  // 1.私有化成员变量，使用private关键字修饰
  // private关键字修饰表示私有的含义，也就是该成员变量只能在当前类的内部使用
  private int id;        // 用于描述学号的成员变量
  private String name;   // 用于描述姓名的成员变量

  // 3.在公有的构造方法中调用set方法进行合理值的判断
  public Student() {}
  public Student(int id, String name) {
    //this.id = id;
    //this.name = name;
    setId(id);
    setName(name);
  }

  // 2.提供公有的get和set方法，并在方法体中进行合理值的判断
  // 使用public关键字修饰表示公有的含义，也就是该方法可以在任意位置使用
  public int getId() {
    return id;
  }
  public void setId(int id) {
    if(id > 0) {
      this.id = id;
    } else {
      System.out.println("学号不合理哦！！！");
    }
  }
```

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

    // 自定义成员方法实现特征的打印
    // 什么修饰符都没有叫做默认的访问权限，级别介于private和public之间
    public void show() {
      //System.out.println("我是" + name + "，我的学号是" + id);
      System.out.println("我是" + getName() + "，我的学号是" + getId());
    }
}
```

JavaBean的概念（了解）

• JavaBean是一种Java语言写成的可重用组件，其它Java 类可以通过反射机制发现和操作这些JavaBean 的属性。

• JavaBean本质上就是符合以下标准的Java类：

 类是公共的

 有一个无参的公共的构造器

 有属性，且有对应的get、set方法

JavaBean，java豆，就是一个封装好的公共类，但里面成员是私有化。就比如水果，就是一个bean。你不能说所有的水果都是红色，但是你拿水果去引用的时候，你可以说苹果是水果，苹果是红色的。西瓜是水果，西瓜是绿色的。如果你说水果是红色的，那就是假命题了。

第八章：static关键字和继承

第一节：static关键字（重点）

基本概念

•使用static关键字修饰成员变量表示静态的含义，此时成员变量由对象层级提升为类层级，也就是整个类只有一份并被所有对象共享，该成员变量随着类的加载准备就绪，与是否创建对象无关。

•static关键字修饰的成员可以使用引用.的方式访问，但推荐类名.的方式。

类层级的意思就好比，人天生只有两只手，只要是人就有两只手，不管对象是老师还是学生还是工人。

使用方式

•在非静态成员方法中既能访问非静态的成员又能访问静态的成员。

(成员：成员变量+ 成员方法，静态成员被所有对象共享)

•在静态成员方法中只能访问静态成员不能访问非静态成员。

(成员：成员变量+ 成员方法，因为此时可能还没有创建对象)

•在以后的开发中只有隶属于类层级并被所有对象共享的内容才可以使用static关键字修饰。(不能滥用static关键字)

构造块和静态代码块（熟悉）
•构造块：在类体中直接使用{}括起来的代码块。
•每创建一个对象都会执行一次构造块。
•静态代码块：使用static关键字修饰的构造块。
•静态代码块随着类加载时**执行一次**。

又见main方法
•语法格式：
public static void main(String[] args){}
•参数使用的举例。

案例题目（重中之重）
•编程实现Singleton类的封装。
•编程实现SingletonTest类对Singleton类进行测试，要求main方法中能得到且只能得到该类的一个对象。

```
/*
    编程实现Singleton类的封装
 */
public class Singleton {

  // 2.声明本类类型的引用指向本类类型的对象，使用private static关键字共同修饰
  //private static Singleton sin = new Singleton();  // 饿汉式
  private static Singleton sin = null;               // 懒汉式

  // 1.私有化构造方法，使用private关键字修饰
  private Singleton() {}

  // 3.提供公有的get方法负责将对象返回出去，使用public static关键字共同修饰
  public static Singleton getInstance() {
    //return sin;
    if(null == sin) {
      sin = new Singleton();
    }
    return sin;
  }
}
12345678910111213141516171819202l
/*
    编程实现Singleton类的测试
 */
public class SingletonTest {

  public static void main(String[] args) {

    // 1.声明Singleton类型的引用指向该类型的对象
    //Singleton s1 = new Singleton();
```

```
    //Singleton s2 = new Singleton();
    //System.out.println(s1 == s2); // 比较变量s1的数值是否与变量s2的数值相等  false
    //Singleton.sin = null;  可以使得引用变量无效
    Singleton s1 = Singleton.getInstance();
    Singleton s2 = Singleton.getInstance();
    System.out.println(s1 == s2); // true
  }
}
```

单例设计模式的概念
•在某些特殊场合中，一个类对外提供且只提供一个对象时，这样的类叫做单例类，而设计单例的流程和思想叫做单例设计模式。

单例设计模式的实现流程
•私有化构造方法，使用private关键字修饰。
•声明本类类型的引用指向本类类型的对象，并使用private static关键字共同修饰。
•提供公有的get方法负责将对象返回出去，并使用public static关键字共同修饰。

单例设计模式的实现方式
•单例设计模式的实现方式有两种：饿汉式和懒汉式，在以后的开发中推荐饿汉式。

第二节：继承（重中之重）

继承的概念
•当多个类之间有相同的特征和行为时，可以将相同的内容提取出来组成一个公共类，让多个类吸收公共类中已有特征和行为而在多个类型只需要编写自己独有特征和行为的机制，叫做继承。
继承的概念
•在Java语言中使用extends(扩展)关键字来表示继承关系。
•如：
public class Worker extends Person{} -表示Worker类继承自Person类
其中Person类叫做超类、父类、基类。
其中Worker类叫做派生类、子类、孩子类。
•使用继承提高了代码的复用性，可维护性及扩展性，是多态的前提条件。
继承的特点
•子类不能继承父类的构造方法和私有方法，但私有成员变量可以被继承只是不能直接访问。
•无论使用何种方式构造子类的对象时都会自动调用父类的无参构造方法，来初始化从父类中继承的成员变量，相当于在构造方法的第一行增加代码super()的效果。
继承的特点
•使用继承必须满足逻辑关系：子类is a 父类，也就是不能滥用继承。
•Java语言中只支持单继承不支持多继承，也就是说一个子类只能有一个父类，但一个父类可以有多个子类。

方法重写的概念
•从父类中继承下来的方法不满足子类的需求时，就需要在子类中重新写一个和父类一样的方法来覆盖从父类中继承下来的版本，该方式就叫做方法的重写（Override）。

方法重写的原则
•要求方法名相同、参数列表相同以及返回值类型相同，从Java5开始允许返回子类类型。
•要求方法的访问权限不能变小，可以相同或者变大。
•要求方法不能抛出更大的异常(异常机制)。

Java开发的常用工具
•文本编辑器（TE，Text Editor）
•记事本、Notepad++、Edit Plus、UltraEdit、...
•集成开发环境（IDE，Integrated Development Environment ）
•Jbuilder、NetBeans、Eclipse、MyEclipse、IDEA、...

又见构造块与静态代码块（笔试）
•先执行父类的静态代码块，再执行子类的静态代码块。
•执行父类的构造块，执行父类的构造方法体。
•执行子类的构造块，执行子类的构造方法体。

第三节：访问控制

修饰符	本类	同一个包中的类	子类	其他类
public	可以访问	可以访问	可以访问	可以访问
protected	可以访问	可以访问	可以访问	不能访问
默认	可以访问	可以访问	不能访问	不能访问
private	可以访问	不能访问	不能访问	不能访问

注意事项
•public修饰的成员可以在任意位置使用。
•private修饰的成员只能在本类内部使用。
•通常情况下，成员方法都使用public关键字修饰，成员变量都使用private关键字修饰。

package语句的由来
•定义类时需要指定类的名称，但如果仅仅将类名作为类的唯一标识，则不可避免的出现命名冲突的问题。这会给组件复用以及团队间的合作造成很大的麻烦！
•在Java语言中，用包（package）的概念来解决命名冲突的问题。
包的定义
•在定义一个类时，除了定义类的名称一般还要指定一个包名，格式如下：
package 包名;
package 包名1.包名2.包名3...包名n;
•为了实现项目管理、解决命名冲突以及权限控制的效果。

定义包的规范
•如果各个公司或开发组织的程序员都随心所欲的命名包名的话，仍然不能从根本上解决命名冲突的问题。因此，在指定包名的时候应该按照一定的规范。
•org.apache.commons.lang.StringUtil
•其中StringUtils是类名而org.apache.commons.lang是多层包名，其含义如下：org.apache表示公司或组织的信息（是这个公司（或组织）域名的反写）；common 表示项目的名称信息；lang 表示模块的名称信息。

包的导入
•使用import关键字导入包。
•使用import关键字导入静态成员，从Java5.0开始支持。

第四节：final关键字（重点）

基本概念
•final本意为"最终的、不可改变的"，可以修饰类、成员方法以及成员变量。

使用方式
•final关键字修饰类体现在该类不能被继承。
-主要用于防止滥用继承，如：java.lang.String类等。
•final关键字修饰成员方法体现在该方法不能被重写但可以被继承。
-主要用于防止不经意间造成重写，如：java.text.Dateformat类中format方法等。
•final关键字修饰成员变量体现在该变量必须初始化且不能改变。
-主要用于防止不经意间造成改变，如：java.lang.Thread类中MAX_PRIORITY等。

常量的概念
•在以后的开发中很少单独使用final关键字来修饰成员变量，通常使用public static final关键字共同修饰成员变量来表达常量的含义，常量的命名规范要求是所有字母都要大写，不同的单词之间采用下划线连。
•public static final double PI = 3.14;

第九章：多态和特殊类

第一节：多态（重中之重）

多态的概念
•多态主要指同一种事物表现出来的多种形态。
•饮料：可乐、雪碧、红牛、脉动、...
•宠物：猫、狗、鸟、小强、鱼、...
•人：学生、教师、工人、保安、...
•图形：矩形、圆形、梯形、三角形、...

多态的语法格式
•父类类型引用变量名= new 子类类型();
•如：
```
Shape sr= new Rect();
sr.show();
```

案例题目
•编程实现Shape类的封装，特征有：横纵坐标，要求提供打印所有特征的方法。
•编程实现Rect类的封装并继承自Shape类，特征有：长度和宽度。
•编程实现ShapeRectTest类，在main方法中分别创建Shape和Rect类型对象并打印特征。

这里的思路是封装，封装三要素：private，constructer，get/set；
继承 extends ，
Shape sr = new Rect(); //•父类类型引用变量名= new 子类类型();

多态的特点
•当父类类型的引用指向子类类型的对象时，父类类型的引用可以直接调用父类独有的方法。
•当父类类型的引用指向子类类型的对象时，父类类型的引用不可以直接调用子类独有的方法。
•对于父子类都有的非静态方法来说，编译阶段调用父类版本，运行阶段调用子类重写的版本*（动态绑定）*。//披着羊皮的狼，羊皮去掉了
•对于父子类都有的静态方法来说，编译和运行阶段都调用父类版本。

引用数据类型之间的转换
•引用数据类型之间的转换方式有两种：自动类型转换和强制类型转换。
•自动类型转换主要指小类型向大类型的转换，也就是子类转为父类，也叫做向上转型。
•强制类型转换主要指大类型向小类型的转换，也就是父类转为子类，也叫做向下转型或显式类型转换。*//比如当父类类型的引用指向子类类 型的对象时，需要调用子类方法，需要做强转。*
•引用数据类型之间的转换必须发生在父子类之间，否则编译报错。

引用数据类型之间的转换
•若强转的目标类型并不是该引用真正指向的数据类型时则编译通过，运行阶段发生类型转换异常。
•为了避免上述错误的发生，应该在强转之前进行判断，格式如下：
if(引用变量instanceof数据类型)
判断引用变量指向的对象是否为后面的数据类型。//意思就是是否是继承关系

多态的实际意义
•多态的实际意义在于屏蔽不同子类的差异性实现通用的编程带来不同的效果。

第二节：抽象类（重点）

抽象方法的概念
•抽象方法主要指不能具体实现的方法并且使用abstract关键字修饰，也就是没有方法体。
•具体格式如下：
访问权限abstract 返回值类型方法名(形参列表);
public abstract void cry();

特征：abstract;没有方法块;

抽象类的概念
•抽象类主要指不能具体实例化的类并且使用abstract关键字修饰，也就是不能创建对象。
无法实例化，但是需要使用，比如今天要吃饭，但是还不知道吃什么，待会走到哪里就吃什么。

抽象类和抽象方法的关系
•抽象类中可以有成员变量、构造方法、成员方法；
•抽象类中可以没有抽象方法，也可以有抽象方法；
•拥有抽象方法的类必须是抽象类，因此真正意义上的抽象类应该是具有抽象方法并且使用abstract关键字修饰的类。

抽象类的实际意义 – 母版，标准。
•抽象类的实际意义不在于创建对象而在于被继承。
•当一个类继承抽象类后必须重写抽象方法，否则该类也变成抽象类，也就是抽象类对子类具有强制性和规范性，因此叫做模板设计模式。

开发经验分享
•在以后的开发中推荐使用多态的格式，此时父类类型引用直接调用的所有方法一定是父类中拥有的方法，若以后更换子类时，只需要将new关键字后面的子类类型修改而其它地方无需改变就可以立即生效，从而提高了代码的可维护性和可扩展型。
•该方式的缺点就是：父类引用不能直接调用子类独有的方法，若调用则需要强制类型转换。

抽象类的应用
•银行有定期账户和活期账户。继承自账户类。账户类中：
•public class Account{
private double money;
public double getLixi(){}
}

第三节：接口（重点）

接口的基本概念
•接口就是一种比抽象类还抽象的类，体现在所有方法都为抽象方法。
•定义类的关键字是class，而定义接口的关键字是interface。
•如：
金属接口货币接口黄金类

练习题目
•编程实现Runner接口，提供一个描述奔跑行为的抽象方法。
•编程实现Hunter接口继承Runner接口，并提供一个描述捕猎行为的抽象方法。
•编程实现Man类实现Hunter接口并重写抽象方法，在main方法中使用多态方式测试。
思路：1、创建一个接口interface Runner 和Hunter。2、同接口类型可以extends继承，其他不行。
3、创建测试类main用多态方式实现。

类和接口之间的关系

名称	关键字	关系
类和类之间的关系	使用extends关键字表达继承关系	支持单继承
类和接口之间的关系	使用implements关键字表达实现关系	支持多实现
接口和接口之间的关系	使用extends关键字表达继承关系	支持多继承

抽象类和接口的主要区别（笔试题）

•定义抽象类的关键字是abstract class，而定义接口的关键字是interface。
•继承抽象类的关键字是extends，而实现接口的关键字是implements。
•继承抽象类支持单继承，而实现接口支持**多实现**。
•抽象类中可以有构造方法，而接口中不可以有构造方法。*//接口只能有抽象方法，所以不能有构造方法*
•抽象类中可以有成员变量，而接口中只可以有常量。*//和抽象方法*

抽象类和接口的主要区别
•抽象类中可以有成员方法，而接口中只可以有抽象方法。
•抽象类中增加方法时子类可以不用重写，而接口中增加方法时实现类**需要重写**（Java8以前的版本）。
•从Java8开始增加新特性，接口中允许出现非抽象方法和静态方法，但非抽象方法需要使用**default关键字修饰**。
•从Java9开始增加新特性，接口中允许出现私有方法。

第十章：特殊类

第一节：内部类（熟悉）

内部类的基本概念
•当一个类的定义出现在另外一个类的类体中时，那么这个类叫做内部类（Inner），而这个内部类所在的类叫做外部类（Outer）。
•类中的内容：成员变量、成员方法、构造方法、静态成员、构造块和静态代码块、内部类。

这里就是类种类。
实际作用
•当一个类存在的价值仅仅是为某一个类单独服务时，那么就可以将这个类定义为所服务类中的内部类，这样可以隐藏该类的实现细节并且可以方便的访问外部类的私有成员而不再需要提供公有的get和set方法。

内部类的分类
•普通内部类-直接将一个类的定义放在另外一个类的类体中。
•静态内部类-使用static关键字修饰的内部类，隶属于类层级。
•局部内部类-直接将一个类的定义放在方法体的内部时。
•匿名内部类-就是指没有名字的内部类。

普通（成员）内部类的格式

•访问修饰符class 外部类的类名{
访问修饰符class 内部类的类名{
内部类的类体;
}
}

普通内部类的使用方式

•普通内部类和普通类一样可以定义成员变量、成员方法以及构造方法等。
•普通内部类和普通类一样可以使用final或者abstract关键字修饰。
•普通内部类还可以使用private或protected关键字进行修饰。
•普通内部类需要使用外部类对象来创建对象。
•如果内部类访问外部类中与本类内部同名的成员变量或方法时，需要使用this关键字。

静态内部类的格式

•访问修饰符class 外部类的类名{
访问修饰符staticclass 内部类的类名{
内部类的类体;
}
}

静态内部类的使用方式

•静态内部类不能直接访问外部类的非静态成员。
•静态内部类可以直接创建对象。
•如果静态内部类访问外部类中与本类内同名的成员变量或方法时，需要使用类名.的方式访问。

局部（方法）内部类的格式

•访问修饰符class 外部类的类名{
访问修饰符返回值类型成员方法名（形参列表）{
class 内部类的类名{
内部类的类体;
}
}
}

局部内部类的使用方式

•局部内部类只能在该方法的内部可以使用。
•局部内部类可以在方法体内部直接创建对象。
•局部内部类不能使用访问控制符和static关键字修饰符。
•局部内部类可以使用外部方法的局部变量，但是必须是final的。由局部内部类和局部变量的声明周期不同所致。

回调模式的概念

•回调模式是指——如果一个方法的参数是接口类型，则在调用该方法时，需要创建并传递一个实现此接口类型的对象；而该方法在运行时会调用到参数对象中所实现的方法（接口中定义的）。

开发经验分享
•当接口/类类型的引用作为方法的形参时，实参的传递方式有两种：
•自定义类实现接口/继承类并重写方法，然后创建该类对象作为实参传递；
•使用上述匿名内部类的语法格式得到接口/类类型的引用即可；

匿名内部类的语法格式（重点）
•接口/父类类型引用变量名= new 接口/父类类型() { 方法的重写};

第二节：枚举（熟悉）

枚举的基本概念
•一年中的所有季节：春季、夏季、秋季、冬季。
•所有的性别：男、女。
•键盘上的所有方向按键：向上、向下、向左、向右。
•在日常生活中这些事物的取值只有明确的几个固定值，此时描述这些事物的所有值都可以一一列举出来，而这个列举出来的类型就叫做枚举类型。

枚举的定义
•使用public static final表示的常量描述较为繁琐，使用enum关键字来定义枚举类型取代常量，枚举类型是从Java5开始增加的一种引用数据类型。
•枚举值就是当前类的类型，也就是指向本类的对象，默认使用public static final关键字共同修饰，因此采用枚举类型.的方式调用。
•枚举类可以自定义构造方法，但是构造方法的修饰符必须是private，默认也是私有的。

枚举类实现接口的方式
•枚举类实现接口后需要重写抽象方法，而重写方法的方式有两种：重写一个，或者每个对象都重写。

第三节：注解（重点）

注解的基本概念
•注解（Annotation）又叫标注，是从Java5开始增加的一种引用数据类型。
•注解本质上就是代码中的特殊标记，通过这些标记可以在编译、类加载、以及运行时执行指定的处理。

注解的语法格式
•访问修饰符@interface 注解名称{
注解成员;
}
•自定义注解自动继承java.lang.annotation.Annotation接口。
•通过@注解名称的方式可以修饰包、类、成员方法、成员变量、构造方法、参数、局部变量的声明等。

注解的使用方式
•注解体中只有成员变量没有成员方法，而注解的成员变量以“无形参的方法”形式来声明，其方法名定义了该成员变量的名字，其返回值定义了该成员变量的类型。
•如果注解只有一个参数成员，建议使用参数名为value，而类型只能是八种基本数据类型、String类型、Class类型、enum类型及Annotation类型。

元注解的概念
•元注解是可以注解到注解上的注解，或者说元注解是一种基本注解，但是它能够应用到其它的注解上面。
•元注解主要有@Retention、@Documented、@Target、@Inherited、@Repeatable。

元注解@Retention
•@Retention 应用到一个注解上用于说明该注解的的生命周期，取值如下：
•RetentionPolicy.SOURCE 注解只在源码阶段保留，在编译器进行编译时它将被丢弃忽视。
•RetentionPolicy.CLASS 注解只被保留到编译进行的时候，它并不会被加载到JVM 中，默认方式。
•RetentionPolicy.RUNTIME 注解可以保留到程序运行的时候，它会被加载进入到JVM 中，所以在程序运行时可以获取到它们。

元注解@Documented
•使用javadoc工具可以从程序源代码中抽取类、方法、成员等注释形成一个和源代码配套的API帮助文档，而该工具抽取时默认不包括注解内容。
•@Documented用于指定被该注解将被javadoc工具提取成文档。
•定义为@Documented的注解必须设置Retention值为RUNTIME。

元注解@Target

- @Target用于指定被修饰的注解能用于哪些元素的修饰，取值如下：

ElementType.ANNOTATION_TYPE	可以给一个注解进行注解
ElementType.CONSTRUCTOR	可以给构造方法进行注解
ElementType.FIELD	可以给属性进行注解
ElementType.LOCAL_VARIABLE	可以给局部变量进行注解
ElementType.METHOD	可以给方法进行注解
ElementType.PACKAGE	可以给一个包进行注解
ElementType.PARAMETER	可以给一个方法内的参数进行注解
ElementType.TYPE	可以给类型进行注解，比如类、接口、枚举

元注解@Inherited

•@Inherited并不是说注解本身可以继承，而是说如果一个超类被该注解标记过的注解进行注解时，如果子类没有被任何注解应用时，则子类就继承超类的注解。

元注解@Repeatable

•@Repeatable表示自然可重复的含义，从Java8开始增加的新特性。

•从Java8开始对元注解@Target的参数类型ElementType枚举值增加了两个：

•其中ElementType.TYPE_PARAMETER 表示该注解能写在类型变量的声明语句中，如：泛型。

•其中ElementType.TYPE_USE 表示该注解能写在使用类型的任何语句中。

常见的预制注解

- 预制注解就是Java语言自身提供的注解，具体如下：

@author	标明开发该类模块的作者，多个作者之间使用,分割
@version	标明该类模块的版本
@see	参考转向，也就是相关主题
@since	从哪个版本开始增加的
@param	对方法中某参数的说明，如果没有参数就不能写
@return	对方法返回值的说明，如果方法的返回值类型是void就不能写
@exception	对方法可能抛出的异常进行说明

常见的预制注解

- 常用的预制注解如下：

@Override	限定重写父类方法, 该注解只能用于方法
@Deprecated	用于表示所修饰的元素(类, 方法等)已过时
@SuppressWarnings	抑制编译器警告