1抽象类

概念

抽象类必须使用abstract class声明
一个抽象类中可以没有抽象方法。抽象方法必须写在抽象类或者接口中。

格式:

```
abstract class 类名{  // 抽象类

}
```

抽象方法

只声明而未实现的方法称为抽象方法（未实现指的是：没有“{}”方法体），抽象方法必须使用abstract关键字声明。

格式:

```
abstract class 类名{  // 抽象类
    public abstract void 方法名() ;    // 抽象方法，只声明而未实现
}
```

不能被实例化

在抽象类的使用中有几个原则:

- 抽象类本身是不能直接进行实例化操作的，即：不能直接使用关键字new完成。
- 一个抽象类必须被子类所继承，被继承的子类（如果不是抽象类）则必须覆写(重写)抽象类中的全部抽象方法。

常见问题

1、 抽象类能否使用final声明?
不能，因为final属修饰的类是不能有子类的 ， 而抽象类必须有子类才有意义，所以不能。
2、 抽象类能否有构造方法?
能有构造方法，而且子类对象实例化的时候的流程与普通类的继承是一样的，都是要先调用父类中的构造方法（默认是无参的），之后再调用子类自己的构造方法。

抽象类和普通类的区别

1、抽象类必须用public或protected修饰(如果为private修饰，那么子类则无法继承，也就无法实现其抽象方法）。默认缺省为 public
2、抽象类不可以使用new关键字创建对象， 但是在子类创建对象时， 抽象父类也会被JVM实例化。
3、如果一个子类继承抽象类，那么必须实现其所有的抽象方法。如果有未实现的抽象方法，那么子类也必须定义为abstract类

2、接口

概念

如果一个类中的全部方法都是抽象方法，全部属性都是全局常量，那么此时就可以将这个类定义成一个接口。
定义格式：

```
    interface 接口名称{
        全局常量 ;
        抽象方法 ;
    }
```

面向接口编程思想

这种思想是接口是定义（规范，约束）与实现（名实分离的原则）的分离。

优点：
1、 降低程序的耦合性
2、 易于程序的扩展
3、 有利于程序的维护

全局常量和抽象方法的简写

因为接口本身都是由全局常量和抽象方法组成 ， 所以接口中的成员定义可以简写：
1、全局常量编写时， 可以省略public static final 关键字，例如：
```
public static final String INFO = "内容" ;
```
简写后：
```
String INFO = "内容" ;
```
2、抽象方法编写时， 可以省略 public abstract 关键字， 例如：
```
public abstract void print() ;
```
简写后：
```
void print() ;
```

接口的实现 implements

接口可以多实现：
格式：
```
class 子类 implements 父接口1,父接口2...{

}
```

以上的代码称为接口的实现。那么如果一个类即要实现接口，又要继承抽象类的话，则按照以下的格式编写即可：
```
class 子类 extends 父类 implements 父接口1,父接口2...{

}
```

接口的继承

接口因为都是抽象部分， 不存在具体的实现， 所以允许多继承,例如：
```
interface C extends A,B{

}
```

注意

如果一个接口要想使用，必须依靠子类。 子类（如果不是抽象类的话）要实现接口中的所有抽象方法。

接口和抽象类的区别

1、抽象类要被子类继承，接口要被类实现。
2、接口只能声明抽象方法，抽象类中可以声明抽象方法，也可以写非抽象方法。
3、接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变量。
4、抽象类使用继承来使用， 无法多继承。 接口使用实现来使用， 可以多实现
5、抽象类中可以包含static方法 ，但是接口中不允许（静态方法不能被子类重写，因此接口中不能声明静态方法）
6、接口不能有构造方法，但是抽象类可以有

3、多态

概念

多态：就是对象的多种表现形式，（多种体现形态）

多态的体现

对象的多态性，从概念上非常好理解，在类中有子类和父类之分，子类就是父类的一种形态 ，对象多态性就从此而来。

ps：方法的重载 和 重写 也是多态的一种， 不过是方法的多态（相同方法名的多种形态）。
重载： 一个类中方法的多态性体现
重写： 子父类中方法的多态性体现。

多态的使用：对象的类型转换

类似于基本数据类型的转换：

- 向上转型：将子类实例变为父类实例
  |- 格式：父类 父类对象 = 子类实例 ；
- 向下转型：将父类实例变为子类实例
  |- 格式：子类 子类对象 = （子类）父类实例 ；

4、instanceof

作用：
    判断某个对象是否是指定类的实例，则可以使用instanceof关键字

格式：
    实例化对象 instanceof 类 //此操作返回boolean类型的数据

5、Object类

概念

Object类是所有类的父类（基类），如果一个类没有明确的继承某一个具体的类，则将默认继承Object类。

例如我们定义一个类：
```
public class Person{
}
```

其实它被使用时 是这样的：
```
public class Person extends Object{
}
```

Object的多态

使用Object可以接收任意的引用数据类型

toString

建议重写Object中的toString方法。 此方法的作用：返回对象的字符串表示形式。
    Object的toString方法， 返回对象的内存地址

equals

建议重写Object中的equals(Object obj)方法，此方法的作用：指示某个其他对象是否“等于”此对象。
    Object的equals方法：实现了对象上最具区别的可能等价关系；也就是说，对于任何非空引用值x和y ，当且仅当x和y引用同一对象（ x == y具有值true ）时，此方法返回true 。

equals方法重写时的五个特性：

    自反性 ：对于任何非空的参考值x ， x.equals(x)应该返回true 。
    对称性 ：对于任何非空引用值x和y ， x.equals(y)应该返回true当且仅当y.equals(x)回报true 。
    传递性 ：对于任何非空引用值x ， y和z ，如果x.equals(y)回报true个y.equals(z)回报true ，然后x.equals(z)应该返回true 。
    一致性 ：对于任何非空引用值x和y ，多次调用x.equals(y)始终返回true或始终返回false ，前提是未修改对象上的equals比较中使用的信息。
    非空性 ：对于任何非空的参考值x ， x.equals(null)应该返回false 。

6、内部类

概念

在Java中，可以将一个类定义在另一个类里面或者一个方法里面，这样的类称为内部类。
广泛意义上的内部类一般来说包括这四种：
1、成员内部类
2、局部内部类
3、匿名内部类
4、静态内部类

成员内部类

成员内部类是最普通的内部类，它的定义为位于另一个类的内部，形如下面的形式：

```
class Outer {
    private double x = 0;

    public Outer(double x) {
        this.x = x;
    }

    class Inner {     //内部类
        public void say() {
            System.out.println("x="+x);
        }
    }
}
```

特点： 成员内部类可以无条件访问外部类的所有成员属性和成员方法（包括private成员和静态成员）。
不过要注意的是，当成员内部类拥有和外部类同名的成员变量或者方法时，会发生隐藏现象，即默认情况下访问的是成员内部类的成员。如果要访问外部类的同名成员，需要以下面的形式进行访问：
外部类.this.成员变量
外部类.this.成员方法

外部使用成员内部类

```
        Outter outter = new Outter();
        Outter.Inner inner = outter.new Inner();
```

局部内部类

局部内部类是定义在一个方法或者一个作用域里面的类，它和成员内部类的区别在于局部内部类的访问仅限于方法内或者该作用域内。

例如：

```
    class Person{
        public Person() {

        }
    }
```

```
class Man{
    public Man(){

    }

    public People getPerson(){
        class Student extends People{   //局部内部类
            int age =0;
        }
        return new Student();
    }
}
```

注意:局部内部类就像是方法里面的一个局部变量一样，是不能有public、protected、private以及static修饰符的。

匿名内部类

匿名内部类由于没有名字，所以它的创建方式有点儿奇怪。创建格式如下：

```
new 父类构造器（参数列表）|实现接口（）
{
 //匿名内部类的类体部分
}
```

在这里我们看到使用匿名内部类我们必须要继承一个父类或者实现一个接口，当然也仅能只继承一个父类或者实现一个接口。同时它也是没有class关键字，这是因为匿名内部类是直接使用new来生成一个对象的引用。当然这个引用是隐式的。

注意:

在使用匿名内部类的过程中，我们需要注意如下几点：

1、使用匿名内部类时，我们必须是继承一个类或者实现一个接口，但是两者不可兼得，同时也只能继承一个类或者实现一个接口。

2、匿名内部类中是不能定义构造函数的。

3、匿名内部类中不能存在任何的静态成员变量和静态方法。

4、匿名内部类为局部内部类，所以局部内部类的所有限制同样对匿名内部类生效。

5、匿名内部类不能是抽象的，它必须要实现继承的类或者实现的接口的所有抽象方法。

6、只能访问final型的局部变量

静态内部类

静态内部类也是定义在另一个类里面的类，只不过在类的前面多了一个关键字static。
静态内部类是不需要依赖于外部类对象的，这点和类的静态成员属性有点类似，并且它不能使用外部类的非static成员变量或者方法.

格式:

```
public class Test {
    public static void main(String[] args)  {
        Outter.Inner inner = new Outter.Inner();
    }
}

class Outter {
    public Outter() {

    }

    static class Inner {
        public Inner() {

        }
    }
}
```

7、 包装类

概述

在Java中有一个设计的原则“一切皆对象”，那么这样一来Java中的一些基本的数据类型，就完全不符合于这种设计思想，因为Java中的八种基本数据类型并不是引用数据类型，所以Java中为了解决这样的问题，引入了八种基本数据类型的包装类。

序号	基本数据类型	包装类
1	int	Integer
2	char	Character
3	float	Float
4	double	Double
5	boolean	Boolean
6	byte	Byte
7	short	Short
8	long	Long

以上的八种包装类，可以将基本数据类型按照类的形式进行操作。
但是，以上的八种包装类也是分为两种大的类型的：
- Number：Integer、Short、Long、Double、Float、Byte都是Number的子类表示是一个数字。
- Object：Character、Boolean都是Object的直接子类。

装箱和拆箱操作

以下以Integer和Float为例进行操作

将一个基本数据类型变为包装类，那么这样的操作称为装箱操作。
将一个包装类变为一个基本数据类型，这样的操作称为拆箱操作，

因为所有的数值型的包装类都是Number的子类，Number的类中定义了如下的操作方法，以下的全部方法都是进行拆箱的操作。

序号	方法	描述
1	public ***byte*** byteValue()	用于Byte->byte
2	public abstract ***double*** doubleValue()	用于Double->double
3	public abstract ***float*** floatValue()	用于Float->float
4	public abstract **int** intValue()	用于Integer->int
5	public abstract ***long*** longValue()	用于Long->long
6	public ***short*** shortValue()	用于Short->short

装箱操作：

在JDK1.4之前 ，如果要想装箱，直接使用各个包装类的构造方法即可，例如：

```
int temp = 10 ; // 基本数据类型
Integer x = new Integer(temp) ; // 将基本数据类型变为包装类
```

在JDK1.5，Java新增了自动装箱和自动拆箱，而且可以直接通过包装类进行四则运算和自增自建操作。例如：

```
Float f = 10.3f ;   // 自动装箱
float x = f ;       // 自动拆箱
System.out.println(f * f) ;     // 直接利用包装类完成
System.out.println(x * x) ;     // 直接利用包装类完成
```

字符串转换

使用包装类还有一个很优秀的地方在于：可以将一个字符串变为指定的基本数据类型，此点一般在接收输入数据上使用较多。

在Integer类中提供了以下的操作方法：

```
public static int parseInt(String s) ：将String变为int型数据
```

在Float类中提供了以下的操作方法：

```
public static float parseFloat(String s) ：将String变为Float
```

在Boolean 类中提供了以下操作方法：

```
public static boolean parseBoolean(String s) ：将String变为boolean
....
....
```

8、可变参数

一个方法中定义完了参数，则在调用的时候必须传入与其一一对应的参数，但是在JDK 1.5之后提供了新的功能，可以根据需要自动传入任意个数的参数。

语法：

```
返回值类型 方法名称(数据类型…参数名称){
        //参数在方法内部 ， 以数组的形式来接收
}
```

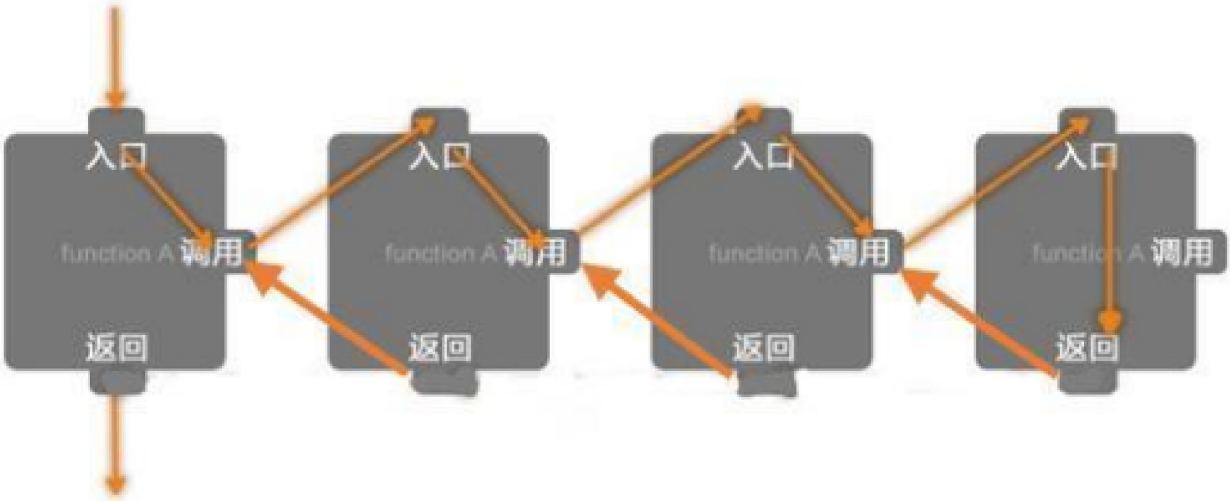
注意：

可变参数只能出现在参数列表的最后。

9、递归

递归，在数学与计算机科学中，是指在方法的定义中使用方法自身。也就是说，递归算法是一种直接或者间接调用自身方法的算法。

递归流程图如下：


入口
function A 调用
返回
入口
function A 调用
返回
入口
function A 调用
返回
入口
function A 调用
返回