# **抽象类**

访问权限修饰符 abstract 类名{}  
不能具体描述，也就是描述不清的类  
关键是用abstract修饰**抽象类不能实例化**  
作用：定义子类的共同特征,所以父类可能没有意义，他只是描述子类的**共同**一部分，所以抽象类用于继承。如果想用抽象类，那么子类必须实现抽象类的所有抽象方法，否则的话子类还只是个抽象类，只能用于给他的子类（孙子类）继承；抽象类不一定有抽象方法

# **抽象方法**

没有{}的方法，他只能声明在抽象类中；  
public abstract void withdraw(String sad);

# **多态性**

对外是一种表现形式，内有多种的具体实现  
一种声明的多种具体实现实现

* 方法重载
* 方法覆盖（重写）
* **多态参数**（重点）

# **编译期类型和运行期类型**

所谓的编译期，可以理解为在写代码时候IDE的自动检验

如

A a = new A();

//其中A就是编译期类型，当调用show()方法的时候就会去A类型中查找相关的方法,也就是说在编译的时候他会去检查A里面是否有show方法

//A()这个构造器类就是运行期类型，在运行的时候他会去构造器中检查是否有show方法

//在这里编译期类型和运行期类型一样

a.show();

当继承时就可以不一样了

* 1-1

A ab =new B();

ab.show();

//可以强制转换为B类

* 父类的引用指向子类的对象  
  **注意:**如果ab对象调用了一个B类型中独有的一个方法是，ab.show()。会产生编译错误，会提示A类中没有这个方法，此时只能强制转换  
  总的来说

# **多态中对成员方法的引用**

* 动态方法编译看左边，运行看右边（左边找父类，右边找子类）
* **静态方法的话，编译运行都看左边**（找父类）
* 成员变量的调用，也是都看左边编译运行都看左边（找父类）

变量不存在被子类复写这个说法

# **多态参数**

### **方法参数多态**

形参是父类类型，传递的实参为任意子类类型  
**作用:**提高代码的扩展性，也就是说子类的类型也可以传递  
比如说：  
public void (String subString)  
也就是说String是父类但是他被许多的子类继承，那么它的子类的可以充当subString的参数。

Account a = new PersonAccount();

publicc void deposit(Account account)

//对于deposit方法来说 a可以传递充当account参数

# **多态对象造型**

* 向上造型 --- 子类的对象父类的类型
* 向下造型 --- 强制转换子类（目的是用子类里的新方法）

# **instanceof操作符**

返回Boolean类型  
result = 对象 instanceof 类型；  
result = a instanceof String;（或者是自己写的类）

# **Object类的作用及地位**

Object是所有类的父类包括数组  
常用方法toString hashCode

方法传递的参数只要是定义为  
public void method(Object o)  
那么传递的参数可以是任意的类型&自己编写的类；  
并且我们每次写的类都是默认继承了Object类的，当你把自己编写的类什么也不写是，创建该类的对象然后调用（输入.）那么他会显示一定的方法，这些方法就是从Object类中继承来的

特别的 有Object[] a 对象型的数组**只能接受引用类型的数组**常规数组不行  
如 int [] a = new int[3];a这样的数组就不行。

# **Object常用方法**

toString() ---将对象转换成字符串返回  
集体过程：getClass().getName()+"@"+integer.toHexS tring(hashcode()) 获得类名@(16位的字符串)  
覆盖方法 public String toString ():  
System.out.print(对象)默认自动调用toString();

equals() --- boolean 比较的是虚地址与==相同(在Object类中)**而在String类中是比较的内容 ==总是比较的虚地址**  
作用：相当于能够重写 ==，而==不能重写

public boolean equals(Object o){

boolean flag = false;

ClassA a = (ClassA)o;

if(title.equals(a.title)&&msg==c.msg){

flag = true;

}

return flag;

}

1. hashCode() --- 返回哈希码16进制

* 如果equals比较返回的是true那么他么的hashCode值一定要相同
* 返回false，hashCode不一定不同
* 覆盖equals，往往需要覆盖hashCode，可以用eclipse自动生成，保证equals返回true，则hashCode相同，返回false，则hashCode不同
* 在Set集合部分有实际应用

# **接口基本语法**

**难点：接口的作用**  
接口相当于一个抽象类，所有方法都是抽象方法的抽象类就是接口  
抽象类中必须写修饰符abstract接口可以省略

【修饰符】 interface 接口名 【extends 父接口列表】{

public static fianl 常量; //只有这两中类型

public abstract 方法; //void read();也是转换成前面的类型

}

# **接口继承接口**

用extends关键字，可以继承多个接口都好分隔（多重继承）

# **接口与抽象类的区别**

| **·** | **抽象类** | **接口** |
| --- | --- | --- |
| 属性 | 不用限制 | 静态常量 |
| 构造方法 | 可有可无 | 没有 |
| 普通方法 | 可以有具体方法 | 必须是抽象方法 |
| 子类 | 单一继承 | 多重继承 |

# **接口的作用 难点**

接口是设计的内容  
面向接口的编程交互在接口中进行

示例:

* 定义一个移动设备必须实现的操作

public interface MobileStorage{

public abstract void readData();

public abstract void writeData();

}

1-2

public interface Player{

void start();

void stop();

}

* 靠接口实现的几个类

public class FlashDisk implements MobileStorage{

@Override

public void readData(){

System.out.println("Using FlashDisk readData")

}

@Override

public void writeData(){

System.out.println("Using FlashDisk writeData")

}

}

public class SDcard implements MobileStorage{

@Override

public void readData(){

System.out.println("Using SDcard readData")

}

@Override

public void writeData(){

System.out.println("Using SDcard writeData")

}

}

**这个实现了多个接口实现**

public class Mp3 implements MobileStorage,Player{

@Override

public void start(){

System.out.println("start")

}

@Override

public void stop(){

System.out.println("stop")

}

@Override

public void readData(){

System.out.println("Using Mp3 readData")

}

@Override

public void writeData(){

System.out.println("Using Mp3 writeData")

}

}

主角（谁用这个接口）以前的思维，通过对象调用方法实现

public class Computer{

private FlashDisk flashDisk;

private SDcard sdcard;

private Mp3 mp3;

public void readDataFrom(FlashDisk d){

d.readData();

}

public void writeDataFrom(FlashDisk d){

d.writeData();

}

}

* **面向接口编程思维**

体现在Computer和MobileStorage的关系，Computer需要移动设备但是不需要具体的，用接口就行了

public class Computer{

private MobileStorage storage; //调用接口

public void setStorage(MobilStorage storage){

this.storage = storage;

}

public void readDataFrom(){

storage.readData();

}

public void writeDataFrom(){

storage.writeData();

}

}

调用

public class Test{

public static void main(String [] arg){

FlashDisk flashDisk = new FlashDisk();

SDcard sdcard = new SDcard();

Mp3 mp3 = new Mp3();

Computer computer = new Computer();

computer.setStorage(flashDisk);

computer.readDataFrom();

computer.writeDataFrom();

//同样的sdcard一样

}

}

接口的拓展性在于如果又有新的内容加进来（硬件）我们直接可以让他继承接口并重写他的方法

# **IO概念**

**概念理解**  
即输入(input)，输出(output)，java中用流来描述输入输出的过程，也就是想流一样持续出入。

* 入还是出事根据内存来说的
  + 读入内存就是输入(input)

把数据读入到内存中执行read操作

* + 从内存中写出就是输出(output)

把数据从内存中写出执行write操作

# **File类**

java.io中的重要类  
**可以表示文件或目录**能对文件或目录进行他们属性的书写，比如说修改日期、大小、名字……需要注意的是他**不能对文件或目录进行读写操作---只有输入输出流才能修改**

### **用法**

* File("文件或目录名字")
* File f1 = new File("aaa.txt")
* File f2 = new File("D:\java\javafile.java")

双斜杠或 反斜杠用一个就行 D:\java == D:/java

常用方法：isexits isFile isDirectory canRead getName getAbslotutePath具体作用请看相关的api

**list()**返回当前目录的子目录及文件所有名字**创建时必须熟String类的数组**

下面是示例并通过for循环取出名字

* 1-1

File f1 = new File("D:\\test");

String [] fileName = f1.list(); //String类型数组

for(String s:fileName)

System.out.println(s);

**listFiles()**返回目录的子目录及文件的所有实例**创建时必须是File类的数组**

下面是示例并通过for循环取出名字

* 1-2

File f1 = new File("D:\\test");

File [] fileName = f1.listFiles(); //File类型数组

for(String s:fileName)

System.out.println(s);

### **list()&listFile接口**

接口FilenameFilter传的是文件名/FileFilter传的是文件  
调用的话就要实现接口重写里面的accept方法

示例：如果文件名字是java.txt就返回true

public class MyFile implement FilenameFilter{

@override

public boolean accept(File dir,String name)

if(name.equal("java.txt")){

return true;

}else{

return false;

}

}

调用：以上面的1-1为例返回名字为java.txt的文件

String [] name = fileName.list(new MyFile());

for(String s:name)

System.out.println(s);

FileFilter也是一样的道理

# **输入流输出流**

* 输入流：往内存里读入

相关的类有Reader类InputStream类及他们的子类如FileInputStream、FileReader

* 输出流：向内存外写出

相关的类有Writer类OutputStream类及他们的子类（对应上面的）

# **字节流及字符流（8位/16位）**

* 字节流对应的就是InputStream和OutputStream 8位 ->1字节

通常用于二进制、视频音频 如：FileInputStream

* 字符流对应的就是Reader和Writer 16位 ->2字节

通常用于文本 如：FileReader

# **节点流和处理流**

* 节点流 封装的数据源，如文件、字符……

如FileReader（File file）当我们要读取File类的文件时如上面的1-2通过FileReader来读，它把file封装了。即把数据源转换成构造器所需要的类型（封装数据源）

File f1 = new File("D:/java.txt");

FileReader fr = null;

try{

fr = new FileReader(f1);

itn i = fr.read();

while(i!=-1){ //read方法规定当返回-1的时候读取结束

System.out.println((char)i)

i = fr.read();

}

}catch(Exception e){

e.printStackTrace();

}finally{

if(fr!= null)

try{

fr.close(); //一定要关闭否则会影响内存读取速率

}catch(Exception e){

e.printStackTrace();

}

}

FileInputStream FileReader

* 处理流 封装的是流对象提供缓冲功能(所以前面一般是Buffered……如BufferedReader),能加快读写效率

类比上面的

FileReader fr = null;

BufferedReader br = null;

try{

br = new BufferedReader(fr); //可以看到他是把FileReader进行了再封装

String line = br.readLine()； //可以看到他读取的是字符串，而不是一个个字符了，所以速度快

// 也可以写在一起

//BufferedReader br = new BufferedReader(new FileReader(fr));

while(line!=null){

System.out.println(line);

line = br.readLine();

}

}catch(Exception e ){

e.printStackTrace();

}

finally{

fr.close();

}

# **字节流字符流的转换 -- 转换流**

InputStreamReader(InputStream in)  
OutputStreamReader(OutputStream out)

示例

BufferedReader br = new BufferedReader(new　InputStreamReader(System.in))；

从键盘中读取数据

# **Sanner 简化了从控制台读取**

Scanner sn = new Scanner(System.in)  
通过nextLine nextString newInt()……获得相应类型的数据

# **IO操作思想**

1. 明确目的
   * 输入 InputStream Reader 读入
   * 输出 OutputStream Writer 写出
2. 操作数据类型
   * 字符流 Reader Writer
   * 字节流 In Out

具体设备

额外功能

* + 高效 利用缓冲区
  + 非高效 不利用
* 需求 -复制 从内存拿出在放进去 text-->textbackup

File f1 = new File("D:/text.txt")

File f2 = new File("D:/textbackup.txt")

BufferedReader br = new BufferedReader(new FileReader(f1));

BufferedWriter bw = new BufferedReader(new FileWriter(f2));

String line = br.readLine(); //line相当于一个中介，把读取的数据存到line，然后把line传递给Write

while(line!=null)

{

bw.writeline(line+'\n');

line = br.readLine();

bw.flush(); // 保证已经操作成功

bw.close();

}