JAVA复习

标识符（ **identifier** ）：计算机中的变量、常量、方法、类等都需要用名字来加以识别的，这种名字就是标识符。标识符是由**字母、数字或符号**组成的字符串，且遵循以下规则：

1、符号只允许**下划线\_**和**美元$符号**

2、不能以数字开头

3、不能使用Java关键字

4、注意英文字母大小写代表不同含义

5、取名应遵循易于理解、便于记忆的原则

例如：合法标识符 **Lotus\_1 wang$**   
非法标识符 **A?2 3ab**

项目名：全部小写，如果项目名是多个英语单词组成，每个单词都要小写

包名：全部小写

类名：首字母大写，也就是帕斯卡命名法，首字母大写，如果类名是多个单词拼成的，每个单词的首字母都要大写

变量名、方法名首字母小写，也就是驼峰命名法，如果名称有多个单词组成，每个单词的首字母都要大写

常量名全部大写

保留字(关键字)就是Java语言中已经被赋予  
特定意义的一些单词

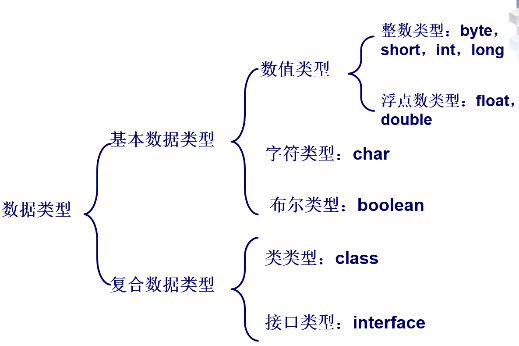
不可以把这类词作为名字来用

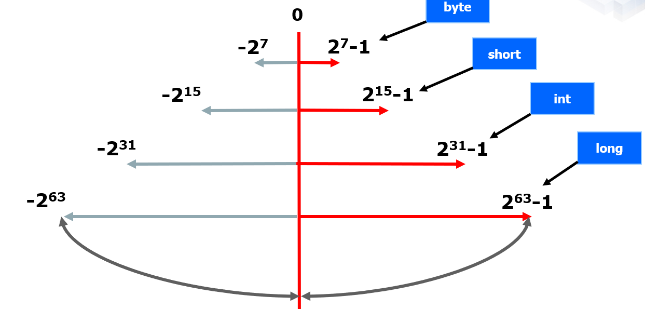
注意：

Java中**true、false、null**都是**小写**，C++中大写

**无 sizeof** 运算符  
Java所有数据类型的长度和表示是**固定**的，与平台无关，不是像在C/C++语言中那样数据类型的长度根据不同的平台而变化。这正是Java语言的一大特点.

**goto**和**const**是Java编程语言中保留的**没有意义**的关键字





5 表示十进制值5

075 表示八进制数值75（也就是十进制数61）

0x9ABC 表示十六进制的数值9ABC（也就是十进制数39612）

整型，如在其后有一个字母“L”表示一个long常量（也可以用小写“l”）

如果一个数值常量中包含小数点或指数部分，或者其后跟有字母F或f（float）, D或d（double），则该数为浮点数.

如果不明确指明浮点数的类型，浮点数缺省为double类型.

3.14159 （double型浮点数）

2.08E25 （double型浮点数）

6.56f （float型浮点数）

float为32位（单精度），double为64位（双精度）

关于字符型

Java中**char**是**16**位的无符号型数据

内存分配给**2个字节**，占16位，最高位不用来表示符号

字符必须用一对单引号括起来，如‘a’，‘B’等

Java采用**Unicode字符集**字符进行编码, 每个编码2字节

与C语言类似，Java也提供转义字符，以反斜杠（\）开头，将其后的字符转变为另外的含义。

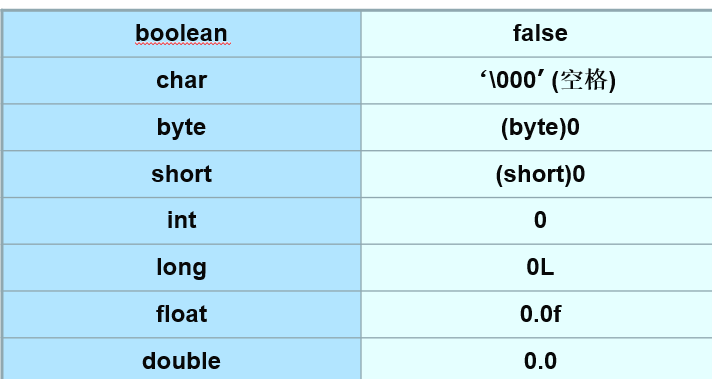
例如：‘A’、 ‘!’ 、‘9’、 ‘好’ 、‘\t’ 、‘怹’、 ‘δ’

逻辑类型

常量：**true ，false**

变量的定义  
使用关键字**boolean**来定义逻辑变量,   
定义时也可以赋给初值：  
boolean x=true, tom=false, jiafei;

**小写**（与C++不同）



复合数据类型：

用户定义的、由一系列简单数据类型及其运算符合而成。

由类类型（**class**）和接口类型（**interface**）组成

例如：String就是由class定义的类类型数据。

各类型所占用的位数从短到长依次为：**(byte,short,char)int long float double**

简单数据类型之间的转换又可以分低级到高级的自动类型转换高级到低级转换



Java语言按运算符对数据的运算结果分类有算术运算符、逻辑运算符、关系运算符、赋值运算符和位运算符；按运算符运算的数据个数分类可分为一元运算符、二元运算符和三元运算符.

算术运算符有五种 加 + 减 – 乘 \* 除 / 取余% 其中除减号可作为一元运算符外均为二元运算符.

另外两个经常使用的一元运算符是

i++ 和 ++i

i-- 和 --i



**顺序结构** 就是按照指令的先后顺序依次执行。

为实现**分支结构**程序设计，Java语言提供了条件分支语句 **if** 和多重分支语句**switch**，根据它们所包含的逻辑表达式的值决定程序执行的方向。

**循环结构**的程序可以对反复执行的程序段进行精炼，用较少的语句执行大量重复的工作。  
Java提供了**for** 、**while**和**do-while**三种循环语句

switch 语句中的表达式的数据类型可以是byte、char、short、int 类型。

根据表达式值与case语句后面的匹配情况决定程序执行的分支

每个case 语句都要有break语句

不匹配的情况执行 default 语句

for语句是三个循环语句中功能最强，使用最广泛的一个。for语句的格式如下：

for (**表达式1; 表达式2; 表达式3**)

循环体语句;

}

表达式1一般是一个赋值语句，它用来给循环控制变量赋初值；

表达式2是一个布尔类型的表达式，它决定什么时候退出循环；

表达式3一般用来修改循环变量，控制变量每循环一次后按什么方式变化。

三个部分之间用“**;** ”分开

for语句的执行过程：

在循环刚开始时，先计算表达式1，在这个过程中，一般完成的是初始化循环变量或其它变量。

根据表达式2的值来决定是否执行循环体。表达式2是一个返回布尔值的表达式，若该值为假，将不执行循环体，并退出循环；若该值为真，将执行循环体。

执行完一次循环体后，计算表达式3。  
在这个过程中一般会修改循环变量。

转入第（2）步继续执行。

* 跳转语句用来实现循环执行过程中的流程转移。在switch语句中使用过的break语句就是一种跳转语句。在Java语言中，有两种跳转语句：**break**语句和**continue**语句.
* Java语言中，可用break和continue**控制循环的流程**
  + break用于强行退出循环，不执行循环中剩余的语句
  + continue则停止执行当前的循环，开始新的循环
* break语句和continue语句都有两种使用形式：
  + 不带标号的break语句和continue语句
  + 带标号的break语句和continue语句
  + 只使用break只能退出内循环；要想达到从内循环体中直接跳转出外循环的目的， 必需与label标号语句连用
* 面向对象思想出现原因及背景：
  + 程序越来越大，系统越来越复杂
  + 编程序需要的人员越来越多，人员交流不方便
* 用**管理者的思维来思考问题**就是使用面向对象的思想。
  + 当做一件事情时，不在考虑怎么做这件事情，而是去考虑使用什么人去做这件事情
* 对象从哪里来？
  + 自己培养一个对象：类的定义
  + 向社会招聘：Java的常用类
* **类：**就是分类标准
  + 例子：公司招聘启事的条件
    - 年龄、性别、籍贯、是否党员（特征）
    - 熟悉Java编程、数据库、Web开发（能力）
  + 两部分构成：
    - 属性（变量、变量）
    - 方法（函数、能力）
* 对象：对象是类的一个实例

例子：公司根据招聘启事招聘的员工

抽象：将具体的实例转化为类

描述出分类标准

将关键信息提取出来并进行建模

**方法的抽象、属性的抽象**

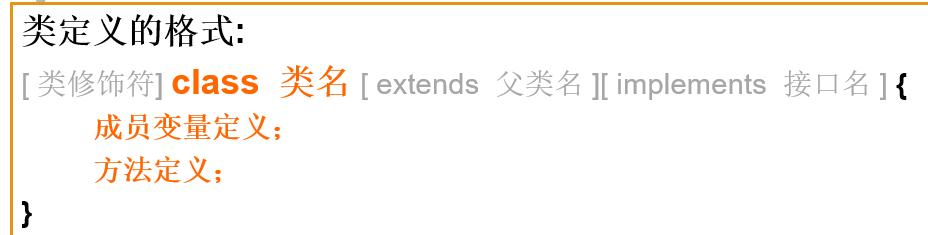
抽象是类定义必须具备的能力，抽象是面向对象思想中最重要的能力。

招聘什么样的人员，最考虑一个管理者的能力

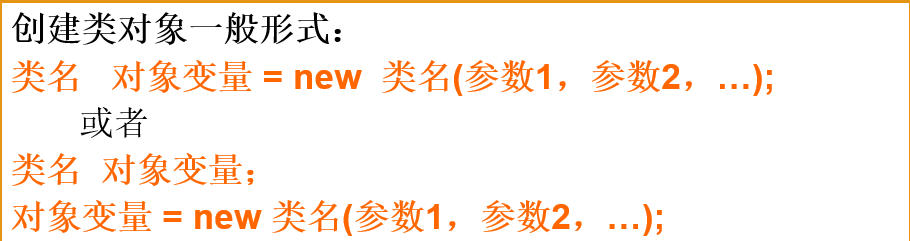
人们能够解决问题的复杂性直接取决于抽象的类型和质量。

在编程之前一定要好好思考，多做分析、多做设计。

* 类的定义包含两部分：
  + 类的说明：由class关键字和类名组成
  + 类的实体：成员变量的定义和成员方法的定义
* 类名的第一个字母要大写，类体要用花括号{}括起来
* 成员变量的定义和成员方法的定义方法和C语言中变量和方法的定义类似。



* 类的实例化-对象的产生
  + 定义对象的引用
  + 采用new关键字实例化对象
  + 对象的使用：
  + 对象产生后可以通过**对象名**和“.”操作符调用对象的变量和方法



构造方法是一个类方法中方法名与类名相同的类方法，  
它的作用是当使用**new关键字**创建一个对象时，自动  
调用并对新建对象进行初始化操作.

构造方法的**特点**

构造方法与类名相同

构造方法没有返回值，不定义返回类型

构造方法可以重载，针对参数数量和类型定义多个同名构造方法

构造方法不能由编程人员显式地直接调用，它是在对象创建时  
由new**运算符自动调用的构造**方法只用来对类实例进行初始化，目的在于简化初始化操作

在Java中，对象的**创建和初始化是统一**的——两者缺一不可

* static修饰的成员变量为类成员变量、也叫静态变量
  + 保存在内存区的公共存储单元中，而不是保存在某个对象的内存区中。任何一个类对象在访问它时都会得到相同的数值。
  + 如果某一个类对象对它进行了修改，所有对象的此成员变量也都做了修改。
  + 可以通过类名调用静态变量
* static修饰的成员方法为类成员方法、也叫静态方法
  + 静态方法属整个类，不用进行实例化操作
  + 静态方法只可以使用类中的静态成员，不能使用非静态成员
  + 可以通过类名调用静态方法
* **继承性:**现实世界中的对象有许多相似的特性，因此在构造类的时候就要写大量重复的代码，导致代码量大且臃肿，而且维护性不高。

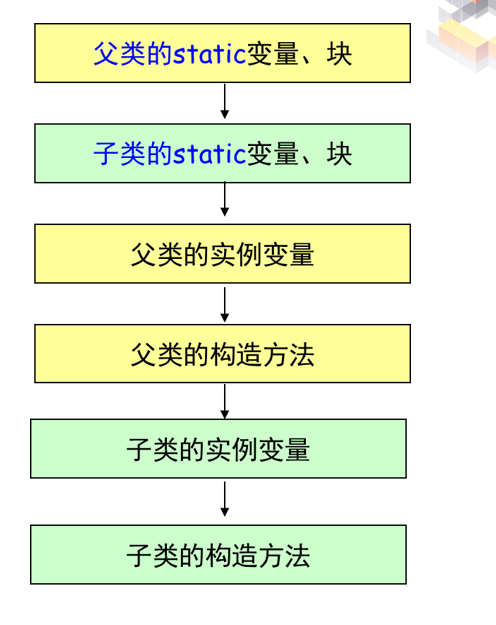
**目的：**减少代码的重复，提高程序可维护性

**方法：**将代码相同的部分提取出来组成一个父类

* 类之间的继承关系是现实世界中遗传关系的直接模拟，它表示类之间的内在联系以及对属性和操作的共享，即子类可以沿用父类（被继承类）的某些特征。子类也可以具有自己独立的属性和操作。
* 子类从父类继承有两个主要的方面：
  1. **属性的继承**
  2. **方法的继承**

****

* **使用extends关键字完成类的继承**
* **Java是一种单继承，一个子类只可以有一个父类，但一个父类可以有多个子类**
* **子类在继承的时候会完成属性的继承和方法的继承**
* **子类比它们的父类具有更多的功能。因为子类是父类的扩展，增加了父类没有的属性和方法**
* **构造方法的继承应遵守以下原则**
  1. **子类可以无条件的继承父类不含参数的构造方法**
  2. **如果子类没有构造方法，则它继承父类无参数的构造方法作为自己的构造方法；  
     如果子类有构造方法，那么在创建子类对象时，则将先执行继承下来的父类的构造方法，然后再执行自己的构造方法**
  3. **对于父类中包含有参数的构造方法，子类可以通过在自己的构造方法中使用super关键字来引用，而且必须是子类构造方法中的第一条语句。**
* 总是先初始化父类的成员；
* 总是先初始化static成员；
* 总是先初始化变量，后初始化方法；



* this关键字总是指向调用该方法的对象
  + 构造器中引用该构造器正在初始化的对象
  + 在方法中引用调用该方法的对象
* this使用方式：
  + 普通直接引用
  + 形参与成员名字重名，使用this区分
  + 引用构造函数
* super关键字指向自己由父类部分生成的对象部分
* super使用方式
  + 普通直接引用，调用父类成员部分
  + 调用父类与子类同名的变量与方法

引用父类构造函数

* 在引用父类的方法时，为了明确的是父类方法需在方法  
  前加**super**。
* 有时，因为继承虽然对子类的方法进行了重新定义，但是当前的方法中的形参明与成员变量名相同或与方法中的局部变量名相同，为了明确其含义，就要采用**this关键字**加以区别。
* 下面的例子中在创建Point3D类对象时引用Point2D类的构造器，使用super关键字来调用父类的方法**super(a,b)**, 对于限制继承的成员变量加上protected 说明，由于Point类和Circle类都存在**toString()**方法，所以在引用**toString()**方时用**this**和**super**加以区别。
* **封装性:**

**基本思想：** 把客观世界中联系紧密的元素及其相关操作组织在一起，使其相互作用隐藏、封装在内部，而对外部对象只提供单一的功能接口

**目的：**将对象的使用者和设计者分开

**类比：**汽车的左转、右转（方向盘 与 车轮）；  
 被陶瓷封装的集成芯片

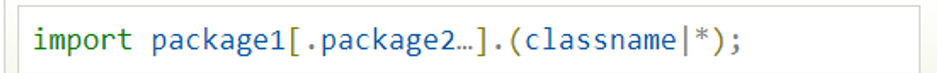
* 所谓**封装**表现在以下几个方面
  1. 在类的定义中设置对对象中的成员变量和方法进行访问的权限。
  2. 提供一个统一供其它类引用的方法。
  3. 其它对象不能直接修改本对象所拥有的属性和方法。
* 对象及成员变量的访问权限
* 允许库创建者声明哪些东西是**程序员**可以使用的，哪些是不可使用的。这种访问控制的级别在“最大访问”和“最小访问”的范围之间.
* 分别包括：public，默认(无关键字)，protected以及private。下面的列表说明访问控制修饰符含义：



* 无论是类或者类成员，只要被声明成**public**，  
  就意味着它可以被从任何地方访问
  + 一个类作为整体对程序的其他部分可见，并不能代表类内的所有属性和方法也同时对程序的其他部分可见，前者只是后者的必要条件，类的属性和方法能否为所有其他类所访问，还要看这些属性和方法自己的访问控制符
  + 实例变量采用**public关键字**，这是一种***非常糟糕***的做法,破坏封装
* 假如类或者类成员不含有一个明确的访问说明，说明它具有默认的访问控制特性。  
  默认的访问控制权规定类或者类成员只能被同一个包中的其他代码所访问，在包外不可见。这种访问特性称为**包访问性**。
* Java语言中的包是类和接口的集合，它从更高级别上提供了对类和接口的存取保护和命名空间管理。
* 使用package关键字可以完成包的定义，放在class文件的第一行，若没有该语句，采用默认的包
  + 包创建了新的命名空间，所以不会与其他包中的命名有冲突



* 使用import关键字导入包，import语句位于 package 语句之后，所有类的定义之前，可以没有，也可以有多条
  + 通配符“\*”，可以代指该包下所有的类



* 用**private**修饰的类成员只能被该类中的其他成员所访问，在该类外不可见。
* 例如，在Point2D类中有两个变量，它们被定义成以下形式：

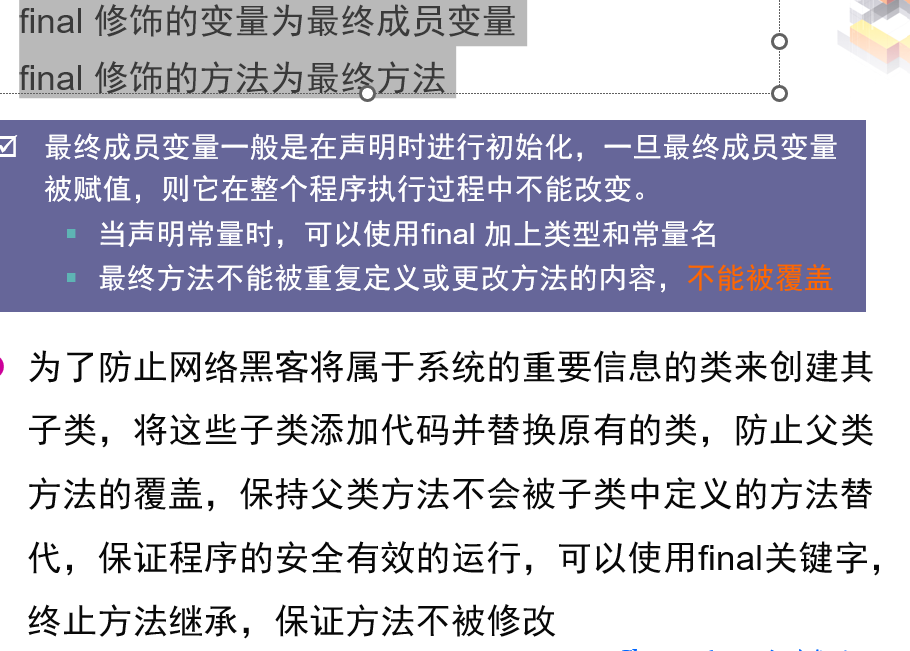
**private double x;**

**private double y;**

* + private（私有的）关键字用来确保可以访问这些实例变量的只能是Point2D类本身的方法。
* 在大多数实际应用的类中，有必要仅仅允许通过方法来对数据进行操作，维护封装。
* 用protected修饰的类成员不但对于**同一个包中  
  的其他类**是可见的，而且对于**其他包中该类的子类也是可见的**。



* 定义:由于**父类和子类可以有同名的方法**，在运行时  
  JVM根据方法的调用对象的不同或者参数个数和类型的不同来查找、决定执行哪个版本的方法，称为**多态性**
* 在程序执行时，JVM对对象某一方法的查找是从该对象类所在层次开始，沿类等级逐级向上进行，把第一个方法作为所要执行的方法。所以，子类的方法可以屏蔽父类的方法
* 多态机制是面向对象程序设计的一个重要特征。
* 多态的特点是采用同名的方式，根据调用方法时调用对象的不同、传送的参数的多少以及传送参数类型的不同，调用不同的方法，这样对于类的编制而言，可以采用同样的方法获得不同的行为特征。
* 多态性
  1. 编译时多态（重载）  
     表现为方法名相同，而参数表不同。  
     典型：System.out.println(); 它的参数有很多种。
  2. 运行时多态（重写）  
     程序运行的时候判断到底是哪个类（父类还是子类的），  
     进而确定实际调用的方法，调用对象不同
* 所谓**方法的重载**，是在类中创建了多个方法，它们具有相同的名称，但有不同的参数和不同的实现。
* 在调用方法时依据其参数个数及类型**自动选择相匹配的方法**去执行。达到各种对象处理类似问题时具有统一的接口目的。
* 可以定义多个构造方法，根据类实例初始化时给定的**参数列表**决定使用哪一个
* 如：**java.util.Vector.remove(int index)**   
   **java.util.Vector.remove(Object o)**
* Java 允许子类对父类的同名方法进行重新定义，因此出现了同名方法而方法的内容不同的情况。  
  出现这种情况时，子类引用方法需要指明引用的父类的方法还是子类的方法。
* 引用父类非私有方法时采用**super**加上成员变量或方法名
  + 如: **super.成员变量**  或 **super.方法名(参数表)**
* 子类覆盖父类中方法时，不能缩小“访问范围”。
  + 如：父类中方法是public的，子类在覆盖时不能改为protected 或private
* 父类的引用可以指向子类的对象，使用父类对象调用被子类覆盖的方法时，可以根据对象的不同调用不同的方法
  + 继承
  + 重写（覆盖）
  + 父类（接口）引用指向子类对象
* 例子：Parent p=new Child();
  + 当使用多态方式调用方法时，首先检查父类中是否有该方法，如果没有，则编译错误；如果有，再去调用子类的同名方法。

+++++++++

* 在面向对象的概念中，**所有的对象都是通过类来描绘的**，  
  但是反过来却不是这样。**并不是所有的类都是用来描绘对象**的，如果一个类中没有包含足够的信息来描绘一个具体的对象，这样的类就是**抽象类**。
* 抽象类往往用来表征我们在对问题领域进行分析、设计中得出的抽象概念，是对一系列看上去不同，但是本质上相同的具体概念的抽象。
* 正是因为抽象的概念在问题领域没有对应的具体概念，所以用以表征抽象概念的抽象类是**不能实例化**的。
* 抽象类是专门设计为子类继承的类，抽象类通常都包括  
  一个或多个抽象方法（**只有方法说明，没有方法体**），抽象方法体中内容，根据继承抽象类的子类的实际情况，有子类完成其抽象方法的代码。
* 定义抽象类的一般形式

***abstract*** class 类名称 {

成员变量；

方法( ){ /\* do something\*/ }； //**定义一般方法**

abstract 方法( )；//**定义抽象方法**

注意无论抽象类和抽象方法都以关键字abstract 开头

* **说明：**
  + 抽象类 里 可以有 非抽象的方法。
  + 抽象类 里 可以没有 抽象方法。
  + 即使没有抽象方法的抽象类也不允许实例化。
  + 一个类里有abstract方法的话，该类必须为abstract。
  + 抽象方法一定不能有方法体（哪怕是空方法体）。
* Java中接口定义的一般形式如下：

[访问控制符] ***interface*** 接口名 {

抽象方法声明

成员变量声明

}

* 从上面的语法规定可以看出，定义接口与定义类非常相似，实际上**可以把接口理解成为一种特殊的抽象类**。
* 接口体现了程序设计的多态性和高内聚低耦合的设计思想。
* 接口不能被实例化。
* 和类的访问级别类似，接口的访问控制符是 public 或者**默认访问**。
  + 如果接口本身定义成public，所有抽象方法和成员变量都是public的，不论是否使用了public访问修饰符
* 接口中的**所有方法**都是抽象方法，在接口中只能给出这些抽象方法的方法名、返回值类型和参数列表，没有方法体。  
  （类似C++的“纯虚类”）
* 接口中可以有成员变量（但变量不能用private和protected修饰）
  + 接口中的变量，本质上都是static，而且是final的 ，无论是否用static修饰，必须以常量值初始化
  + 实际开发中，常在接口中定义常用的变量，作为全局变量使用
  + 访问形式： 接口名.变量名
* **一个接口不能继承其它的类，但是可以继承别的接口**

**一个类可以实现多个接口**

* 一旦接口被定义，一个或多个类可以实现该接口。  
  为了实现一个接口，在类定义中使用**implements关键字**。其一般形式如下：

[访问控制符] class 类名 [extends 父类名]

[implements 接口名1[，接口名2…]] {

成员变量声明

成员方法声明

}

* 同子类对象可以赋给父类引用一样，**接口引用变量可以  
  指向实现对象**。
  + 可以定义一个接口类型的引用变量，并将任何实现了该接口的类的对象赋给它。当通过接口引用变量调用方法时，将根据**动态绑定**的原则来决定方法的正确调用。
  + 需要注意的是，虽然实现对象可以赋给接口引用，但该引用只能访问被它的接口定义声明的内容，而隐藏实现对象中的特殊内容。
* 从语法定义层面
  + 使用abstract class的方式定义Demo抽象类的方式：

abstract class Demo {

abstract void method1();

abstract void method2();

method2();

…

}

* + 使用interface的方式定义Demo抽象类的方式：

interface Demo {

void method1();

void method2();

…

}

* 从编程的角度
  + 抽象类在Java语言中表示的是一种继承关系，一个类只能使用一次继承关系。一个类却可以实现多个接口。
  + 在抽象类的定义中，我们可以赋予方法的默认行为。  
    但是在接口的定义中，**接口却不能拥有默认行为**。
  + 接口的实现只要有相关的函数名称就可以了。即便没有任何实现都可以。
  + **如果一个接口中定义的方法名改变了，那么所有实现此接口的子类显然将无法通过编译**
* 从设计层面  
  接口比抽象类好，一般情况下，如果能用接口就不用抽象类。
  1. 从设计上讲和抽象类有很大不同。
     + 接口表示实现类**尊崇接口的“协议”**，并不是接口的特征，接口既然定义了就不能随便修改。所以设计上，接口不应该很“大”。（后面的线程实现就是例子）
     + 如果一个子类继承了抽象类就决定了这个类的主要特征。

Java常用类：

* 字符串处理类
  + String类
* 基本数据类型包装类
  + Byte、Integer、Short 、Long、Float、Double、Character
* 数学计算类
  + Math、BigInteger 、Random
* 日期处理类

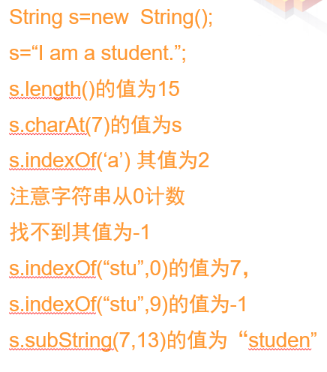
Date、Calendar

* 字符串是**字符组成的序列**，是编程常用的数据  
  类型。字符串分为常量和变量。
  + 字符串常量是指其值保持不变的量，使用双引号括起来的字符序列。
  + 字符串变量要通过String 类来实现，需要声明和初始化。
  + String类在java.lang包中，java.lang包中的类被默认引入，可以直接使用。

**声明格式： String 字符串变量；**

**初始化： 字符串变量 = new String（）；**

**合并使用 String 字符串变量 = new String（）；**



**运算符‘+’的作用是将前后两个字符串连接起来**

字符串操作：(字符串有关的函数)

1. length() 返回字符串长度
2. char charAt(int index)  
   返回字符串中第index个字符
3. int indexOf(int ch)  
   返回字符串中字符ch第一次出现的位置
4. int indexOf(String str,int index)返回值为，从字符串的第index位置开始，子串str第一次出现的位置
5. subString(int index1 ,int index2)返回的是从字符串的第index1位置开始到index2位置结束的子串
6. String[] split(String regex)，根据传入的参数字符串分割字符串

异常处理：

* 异常 指程序运行过程中出现的非正常现象，例如用户输入错误、除数为零、需要处理的文件不存在、数组下标越界等。
* 由于异常情况总是难免的，良好的应用程序除了具备用户所要求的功能以外，还应该具备预见并处理可能发生的各种异常的功能。这种对异常情况进行处理的技术成为**异常处理**。
* 计算机系统对异常处理通常由两种方法
  + 第一种是程序直接检测程序中的错误，遇到错误给出错误信息并终止程序的运行。
  + 第二种办法是由程序员在程序中加入异常处理功能

异常处理的处理机制：

* 程序设计时必须考虑发生的突发错误，避免程序被中断或破坏，在以前的程序设计语言中，通常通过if-else结构来手工判断，不仅增加的程序员的工作，而且会遗漏隐藏的错误。
* Java提供了功能强大的异常处理机制，可以方便地
  + 在程序中监视可能发生异常的程序块
  + 并将所有异常处理代码集中放置在程序的某处

使完成正常功能的程序代码与进行异常处理的程序代码分开。

* Java中处理异常两种方法：
  + 在发生异常的地方直接处理

将异常抛给调用者，让调用者处理

* 在Java的异常处理机制种引进了很多用来描述和处理  
  异常的类，成为异常类。每个异常类反映一类运行错误，  
  类定义中包含了该类异常的信息和对异常进行处理的方法
  + 程序执行中如出现异常，系统会检测到并自动生成一个相应的异常类对象，然后交给运行时系统
  + 运行时系统自动寻找相应的异常处理代码处理这一异常。若找不到可以处理该异常的代码，则运行时系统将终止，程序运行将推出。
* Java将异常分为Exception（异常）和Error（错误）两大类。
  + **Exception类** 解决由程序本身及环境所产生的异常，
  + **Error类** JVM本身错误，不能被程序员处理。
  + Exception类异常可以被捕获并进行处理，而对Error类异常，程序员通常无能为力，只能在其发生时由用户按照系统提示关闭程序。

异常分类：

* 非检查性异常：Error 和 RuntimeException 以及子类。 javac在编译时，不会提示和发现这样的异常，我们可以编写代码处理（使用try…catch…finally）这样的异常，也可以不处理。这样的异常发生的原因多半是代码写的有问题。
  + 如除0错误：ArithmeticException
  + 错误的强制类型转换错误：ClassCastException
  + 数组索引越界:ArrayIndexOutOfBoundsException
  + 使用了空对象:NullPointerException

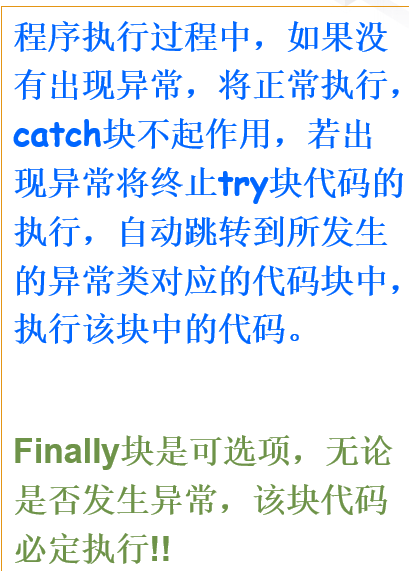
检查性异常：除了Error 和 RuntimeException的其它异常。javac强制要求程序员为这样的异常做预备处理工作（使用try…catch…finally或者throws），否则编译不通过。

* + 打开不存在的文件：FileNotFoundException

异常处理的方法：

* 异常处理有两种方法
  + 使用try…catch…finally结构

通过throws 和 throw 抛出异常

* try…catch…finally一般形式

**try**  {

可能出现异常的程序代码

} **catch**(异常类名 1 异常对象名1) {

异常类名1 对应的异常处理代码

} **catch**(异常类名2 异常对象名2) {

异常类名2对应的异常处理代码

**}**

…

**finally** {

无论是否发生异常，一定会执行的代码

}

* 语法是依据下列的顺序来处理异常
  + **try** 程序块若是有异常发生时，程序的运行便中断，并抛出“异常类所产生的对象”
  + 抛出的对象如果属于**catch()**括号内欲捕获的异常类，则**catch**会捕捉此异常，然后进到**catch**的块里继续运行
  + 无论**try** 程序块是否有捕捉到异常，或者捕捉到的异常是否与**catch()**括号里的异常相同，最后一定会运行**finally**块里的程序代码

抛出异常：

* 如果类中定义的方法本身不捕获某种异常，将该种异常的捕获和处理交给调用它的方法，这是需要在声明本方法时，使用throws关键字抛出**多个异常**，其方法定义具体格式为：

[修饰符] 返回值类型 方法名 [（参数表）] throws 异常类型名 {

声明部分

语句部分

}

* + **这里指给出异常类型名，而不列追加信息**

**通常直接由Java虚拟机处理**

//**在自己编写的方法中主动抛出异常**

//**方法本身不对异常捕获和处理，而由调用它的方法去处理**

class MyMath{

public int div (int i, int j)  **throws Exception** { **//或ArithmeticException**

return (i / j) ;

}

}

**主类**

public class ExceptionDemo03{

public static void main(String args[]){

MyMath m = new MyMath() ;

**try{**

int temp = m.div(10, 0) ;

System.out.println(temp) ;

**}catch** (Exception e){

System.out.println("捕获 除数为零异常") ;

**e.printStackTrace(); // 打印异常**

**}**

**throw new Exception("抛着玩的。") 人为抛出**

}

}

自定义异常类：

* 自定义异常类可以通过继承Exception类来实现，  
  一般格式为：

**class 自定义异常类名 extends Exception {**

**异常类体**

**}**

* 程序员可以在程序中通过throw语句抛出异常，其格式为：

**throw new 异常类名（信息）**

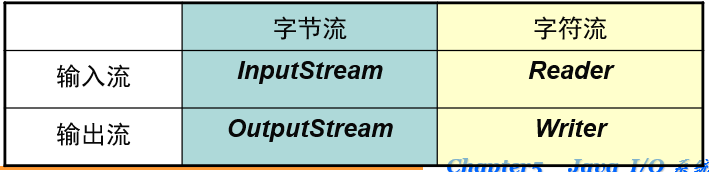
* + 异常类名可以选择系统异常类名，也可以使用自定义异常类名
  + 信息项可选，若有，该信息增加在toString()方法的返回值中

输入输出类库：

* Java的输入输出是以流（stream)的方式进行处理的。流是在计算机的 输入、输出操作中流动的数据序列。  
  Java 按流的单位分有位流（字节流）和字符流；按流动方向分为输入流和输出流。

I/O流的分类：

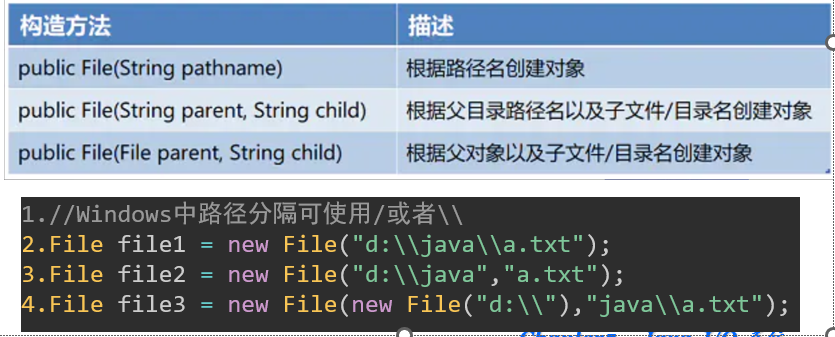
* 按所读写的数据类型分两类：
  + **字节流类**（Byte Streams） 字节流类用于向字节流读写单个字节（8 bit）。一般地，字节流类主要用于读写诸如图象或声音等的二进制数据。（面向计算机）
  + **字符流类**（Character Streams） 字符流类用于向字符流读写一个字符（一个字符根据编码的不同，对应的字节也不同）。字符流类用来处理文本文件。（面向人类）
  + Java SDK所提供的所有流类型位于java.io包内部，  
    全部继承自以下**四种抽象流类型**。



文件处理：

* 在程序中要对磁盘文件或目录进行操作，首先要对文件或目录建立连接，为此Java提供了File类。
* File类位于java.io包中，但不是流类，它不负责输入或输出，而专门用来管理磁盘文件和目录。

类的构造方法



File类常用方法：



File类——注意事项：

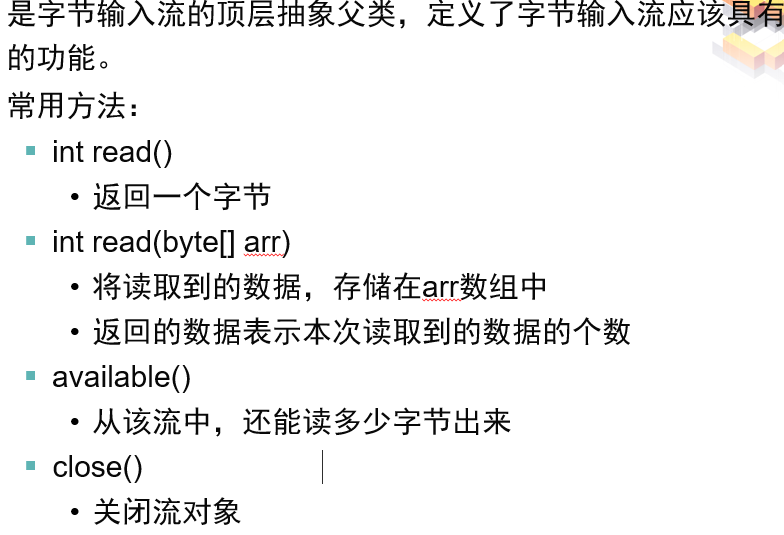
* delete()方法在删除目录时如果其内有内容则无法删除，需要先清空目录下的内容，再删除目录本身
* isDirectory()和isFile()方法在判断是否为文件或者目录时，如果文件或目录不存在则返回false
* createNewFile()方法在创建文件时，如果文件所在的目录不存在则创建失败并抛出异常
* String[] list()和 File[] listFiles() 都是返回对象包含的所有的文件和目录，返回类型不同

字节流：

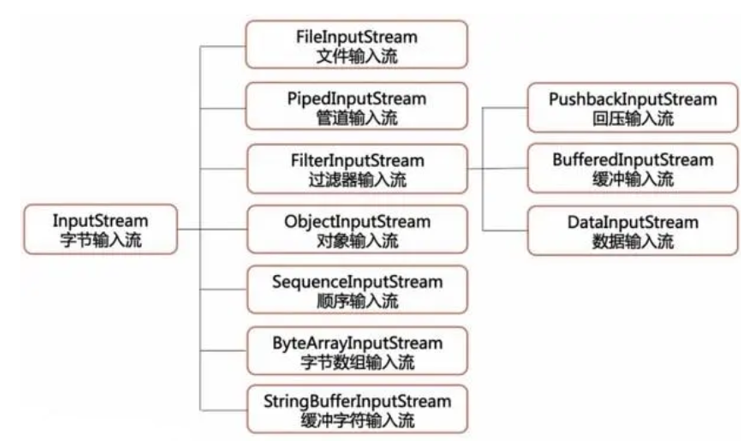
* 可以直接处理字节信息的流对象
* 计算机中一切数据都是字节数据，无论是文字、音频、图片、视频、网页等内容，底层都是字节数据
* 字节流可以操作计算机中一切数据
* 所有其他流对象，底层都需要依赖字节流
* 顶层的抽象类：
  + InputStream

OutputStream

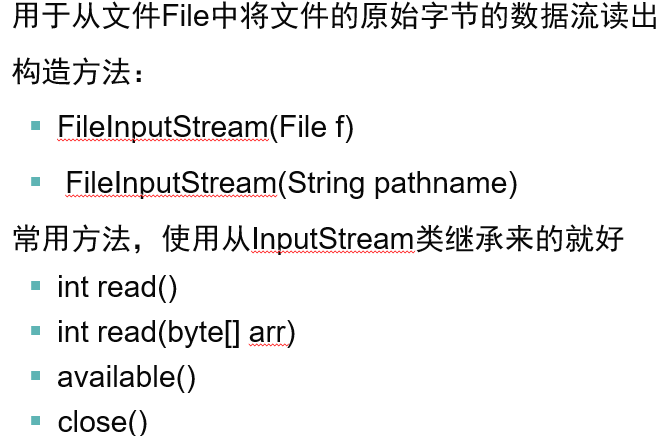
Inputstream类：



IuputStream子类：



FileInputStream:

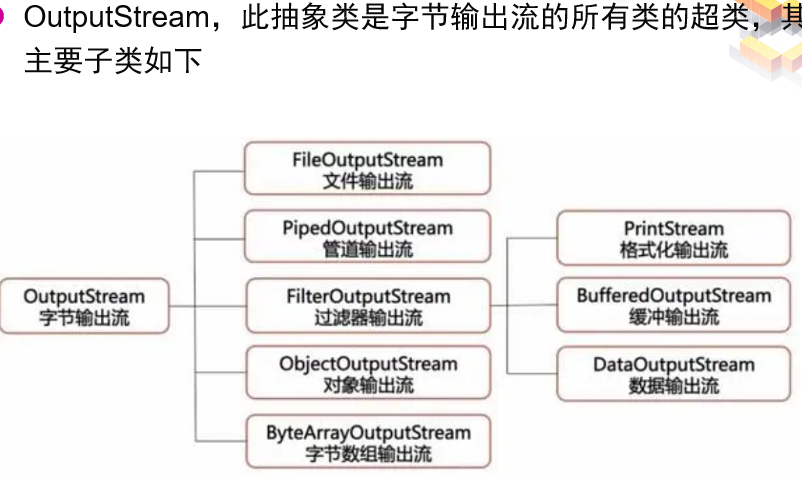


OutputStream:

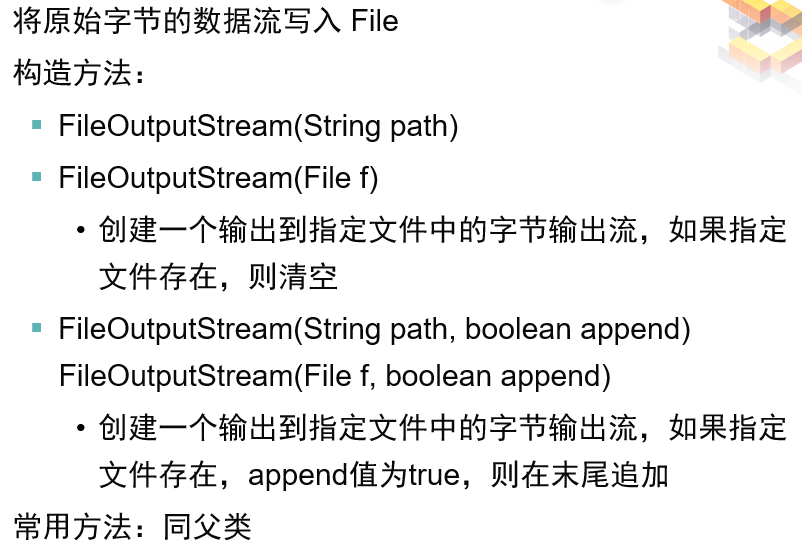
* 是字节输出流的顶层抽象父类，至少应该有一个写出一个字节的方法
* 常用方法：
  + write(int b)
    - 将一个字节存储到目标文件中
  + write(byte[] arr)
    - 将指定的字节数组中的全部字节，写出到目标文件中
  + write(byte[] arr, int offset, int len)
    - 将指定的字节数组的一部分，写出到目标文件中
    - 从offset索引开始，一共len个字节
  + close()

关流

OutputStream子类：



FileOutputStream:



字符流：

* 字符流用于直接操作字符的流对象
* 字符流是在字节流的基础上实现的，字符流将字节流按照相应的编码方式转化为字符流，从而变成人类可以读懂语句
* 字节流面向计算机，字符流面向人类
* 顶层抽象类
  + Reader

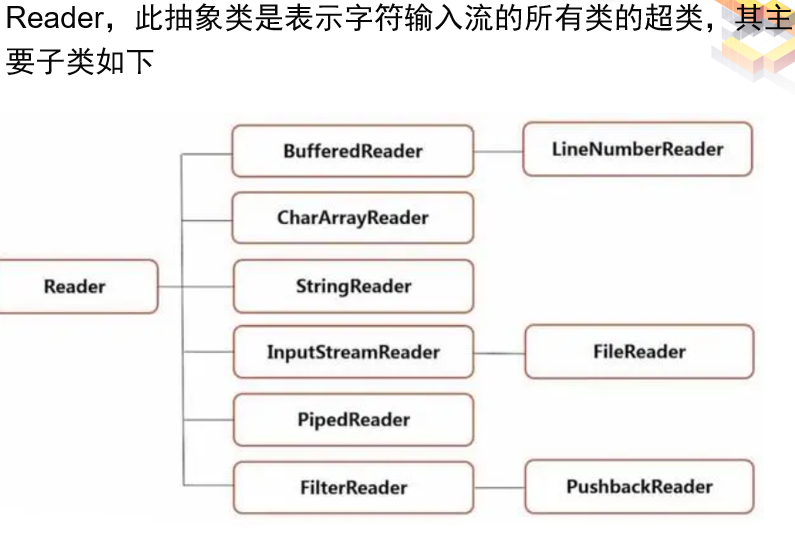
Writer

Reader类：

* 用来以字符方式从流中读入数据。 Reader类中包含了  
  一套所有字符输入流都需要的方法，可以完成最基本的从字符输入流读取数据的功能。
* 常用方法：
  + int read()
    - 读取一个完整的字符，如果返回-1表示到达文件末尾
  + int read(char[] chs)
    - 读取一系列字符到一个字符数组中
  + close()

关闭流对象

Reader子类：



InputStreamreader:

* InputStreamReader：字节流转字符流，它使用的字符集可以由名称指定或显式给定，否则将接受平台默认的字符集。
* 构造方法：
  + InputStreamReader(InputStream in)
  + InputStreamReader(InputStream in, Charset cs)
  + InputStreamReader(InputStream in, CharsetDecoder dec)
  + InputStreamReader(InputStream in, String charsetName)
* 特有方法：
  + String getEncoding()

FileReader:

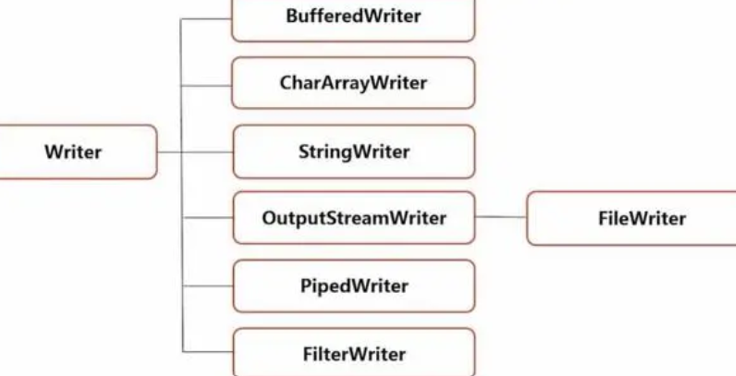
* **按照字符的方式，从文件读取信息**
* **构造方法：**
  + FileReader(String path)
  + FileReader(File f)
    - 将路径封装成字符文件输入流对象
* **常用方法：同父类**
  + int read()
    - 读取一个完整的字符，如果返回-1表示到达文件末尾
  + int read(char[] chs)
    - 读取一系列字符到一个字符数组中
  + close()
    - 关闭流对象

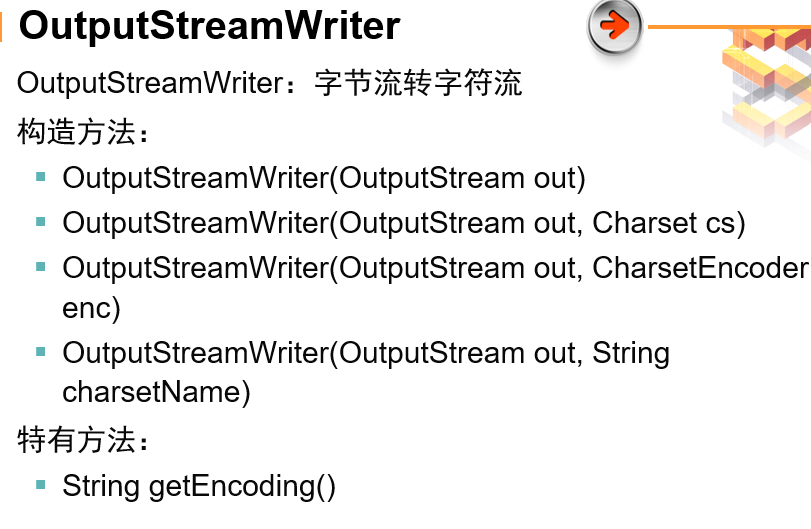
Writer类：

* 用来**以字符方式**向输出流中写入数据
* 常用方法：
  + writer(int c) 将一个字符写出到目标文件
  + write(char[] chs) 将一个字符数组写出到目标文件
  + write(char[] chs, int offset, int len) 将一个字符数组中的从offset开始一共len个字符写出到目标文件
  + write(String str) 写出一个字符串
  + write(String str, int offset, int len) 将字符串的一部分写出
  + flush() 刷新缓冲区

close() 关闭流对象

Writer子类：





FileWriter:

* 用来写入字符文件的类
* 构造方法：
  + FileWriter(String path)
  + FileWrtier(File f)
    - 将一个路径封装成字符输出流，如果文件存在，则先清空
  + FileWriter(String path, boolean append)
  + FileWriter(File f, boolean append)
    - 将一个路径封装成字符输出流，如果第二个参数为true，则在文件存在的情况下，可以追加
* 常用方法：同父类

带缓冲区的字符流:

* 包装类，将没有缓冲功能的普通字符流，包装加强之后形成高效的缓冲字符流包装类也是顶层抽象父类的子类
* BufferedReader构造方法：
  + BufferedReader(Reader r)
  + BufferedReader(Reader in, int sz)
* BufferedWriter构造方法
  + BufferedWriter(Writer w)
  + BufferedWriter(Writer out, int sz)
* BufferedReader特有方法：
  + String readLine()
* BufferedWriter特有方法：

void newLine()

基本流:

* 为了减少程序开发人员，因频繁应用标准的输入输出  
  设备，需要频繁地建立输入输出流对象的工作量，java  
  系统**预先定义好3个流对象**，分别表示标准输出设备、标准输入设备和标准错误设备。他们分别是：
  + System.in ：用于程序的输入； 对应外设为键盘
  + System.out：用于一般输出； 对应外设为屏幕
  + System.err：用于显示出错信息； 对应外设为屏幕
* System 类的所有属性都是**静态static**的，调用时以类名System为前缀。  
  **上述3个流对象均为静态属性。**