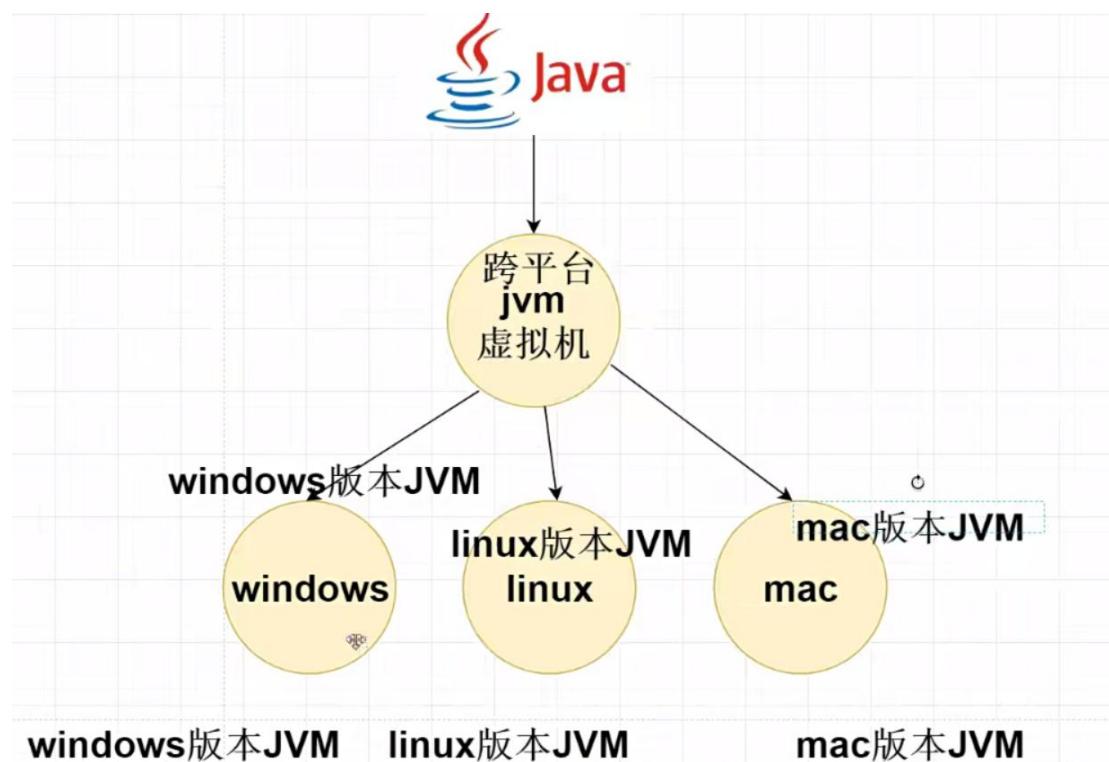


Java 跨平台原理：安装一个与操作系统对应的 jvm 即可。jvm 相当于一个翻译。



- **Java核心机制-Java虚拟机 [JVM java virtual machine]**
- ✓ 基本介绍
- 1) **JVM是一个虚拟的计算机，具有指令集并使用不同的存储区域。负责执行指令，管理数据、内存、寄存器，包含在JDK中。**
- 2) **对于不同的平台，有不同的虚拟机。**
- 3) **Java虚拟机机制屏蔽了底层运行平台的差别，实现了“一次编译，到处运行” [说明]**

Jdk、jre、jvm 的区别和联系：https://blog.csdn.net/Merciful_Lion/article/details/121750939

jdk 目录：

bin：存放一些可执行程序如 javac。

Include：由于 jdk 是由 C 和 C++ 实现的，启动时需要引入一些 C 的头文件，存放在该目录中。

再说三个lib目录：

JDK下的lib包括java开发环境的jar包，是给JDK用的，例如JDK下有一些工具，可能要用该目录中的文件。例如，编译器等。

JRE下的lib只是运行java程序的jar包，是为JVM运行时候用的。包括所有的标准类库，和扩展类。

JDK下的JRE下的lib是开发环境中，运行时需要的jar包。最典型的就是导入的外部驱动jar包。因为编译时，系统找的是jdk下的jre。而不是最外层的jre。

数据类型：

数据类型	默认值	大小
boolean	false	1比特
char	'\u0000'	2字节
byte	0	1字节
short	0	2字节
int	0	4字节
long	0L	8字节
float	0.0f	4字节
double	0.0	8字节

因为 java 采用 Unicode 字符集而不是 ASCII 字符集，所以一个 char 占用两个字节。
Boolean 没有说明占多少空间，取决于 jvm。

Java 开发注意事项：

一个源文件最多包含一个 public，其他类的个数不限。

文件名必须和 public 类名相同。

转义字符：

\r 表示回车，将光标放到所在行起始位置。

+号的使用：

左右两边有一个是字符串时做拼接运算。

编码：

- 介绍一下字符编码表 [sublime 测试]

ASCII (ASCII 编码表 一个字节表示，一个128个字符，实际上一个字节可以表示256个字符,只用128个)
Unicode (Unicode 编码表 固定大小的编码 使用两个字节来表示字符，字母和汉字统一都是占用两个字节，这样浪费空间)

utf-8 (编码表，大小可变的编码 字母使用1个字节，汉字使用3个字节)

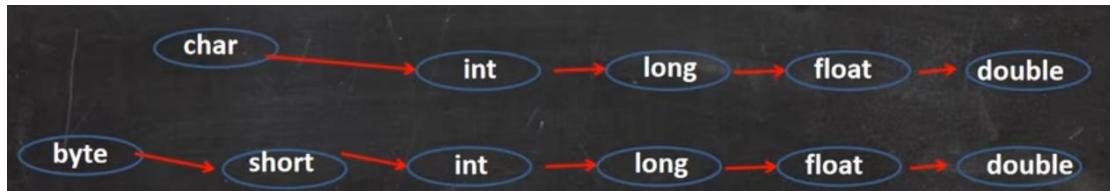
gbk (可以表示汉字，而且范围广，字母使用1个字节，汉字2个字节)

gb2312 (可以表示汉字，gb2312 < gbk)

big5 码(繁体中文, 台湾, 香港)



自动类型转换：



char 和 byte、short 变量不能自动转换。但是可以强制转换。

常量可以自动转换。如 char a = 97; 这个 97 是 int 型。

1. 有多种类型的数据混合运算时，系统首先自动将所有数据转换成容量最大的那种数据类型，然后再进行计算。

```

int num2 = 10;
//float f = num2 + 1.2;
double d2 = num2 + 1.2;
  
```

2. 当我们把精度(容量)大的数据类型赋值给精度(容量)小的数据类型时，就会报错，反之就会进行自动类型转换。

3. (byte, short) 和 char 之间不会相互自动转换。

```

byte b = 10;
char c = b;
double d3 = 100;
int num3 = 1.1;
  
```

4. byte, short, char 他们三者可以计算，在计算时首先转换为 int 类型。

5. boolean 不参与转换 即使 byte、short、char 单独运算，也是先转换为 int

6. 自动提升原则：表达式结果的类型自动提升为操作数中最大的类型



3. char类型可以保存 int的常量值，但不能保存int的变量值，需要强转

```
char c1 = 100;
int m = 100;
char c2 = m;
char c3 = (char)m;
System.out.println(c2);
```

4. byte和short类型在进行运算时，当做int类型处理。

关系运算符：

InstanceOf：检查运行时的对象是否是类的对象或是类的子类的对象。

赋值运算符：

复合赋值运算符会进行类型转换：

```
byte b = 3;
```

b += 2; 等价于： b = (int) (b + 2);

Switch 执行流程：

- switch注意事项和细节讨论

//SwitchDetail.java

1. 表达式数据类型，应和case 后的常量类型一致，或者是可以自动转成可以相互比较的类型，比如输入的是字符，而常量是 int
2. switch(表达式)中表达式的返回值必须是：(byte,short,int,char,enum[枚举],String)

```
double c = 1.1;
```

```
switch(c){//错误
```

```
    case 1.1 : //错误
```

```
        System.out.println("ok3");
```

```
        break;
```

3. case子句中的值必须是常量，而不能是变量

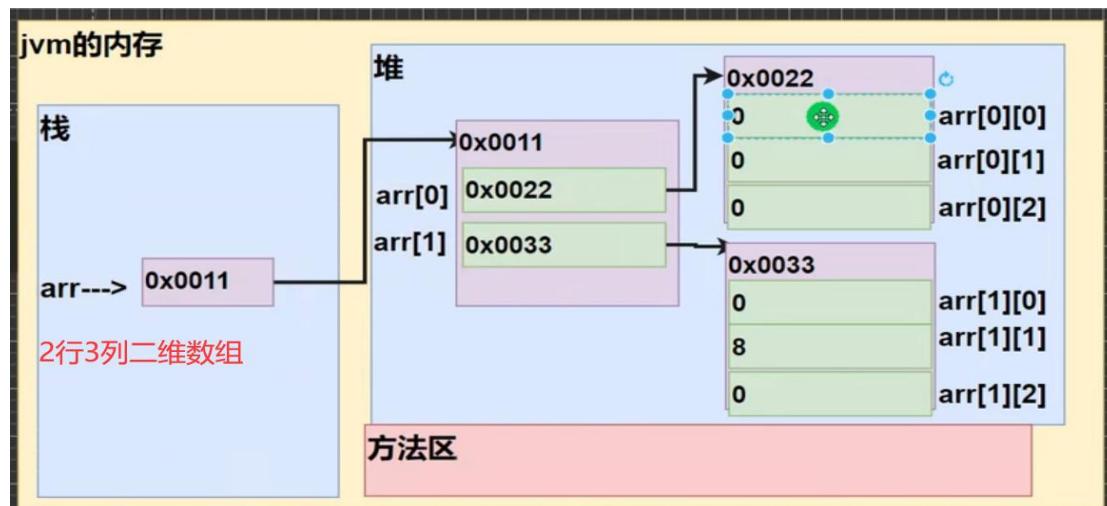
4. default子句是可选的，当没有匹配的case时，执行default

5. break语句用来在执行完一个case分支后使程序跳出switch语句块；如果没有写break，程序会顺序执行到switch结尾

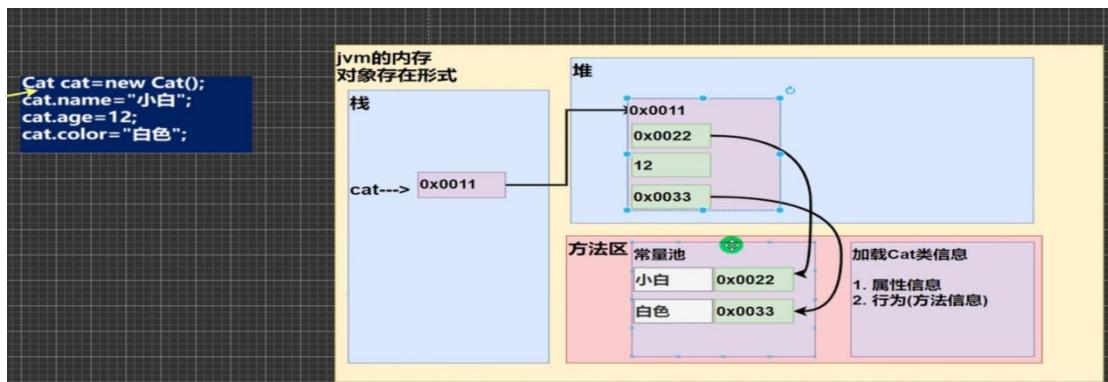
二维数组：

初始化时可以省略列数：

```
int[][] arr = new int[2][];
```



类和对象：

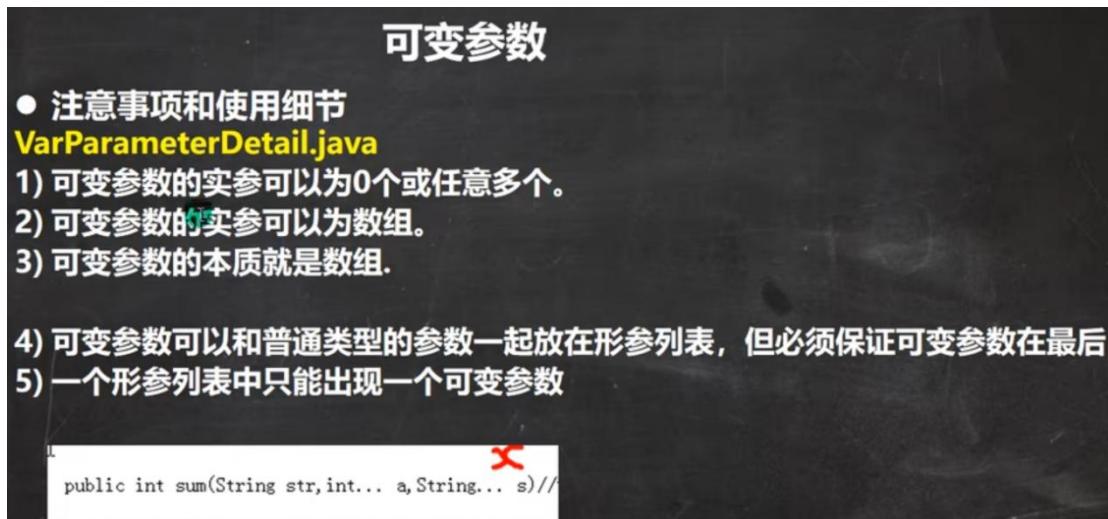


方法重载：

方法名相同，参数列表(参数类型、个数、顺序至少一个)不同，返回类型无要求，参数名无要求。

返回类型不同，其他相同的，不是重载，是方法的重复定义，编译会失败。

可变参数：



全局变量和局部变量：

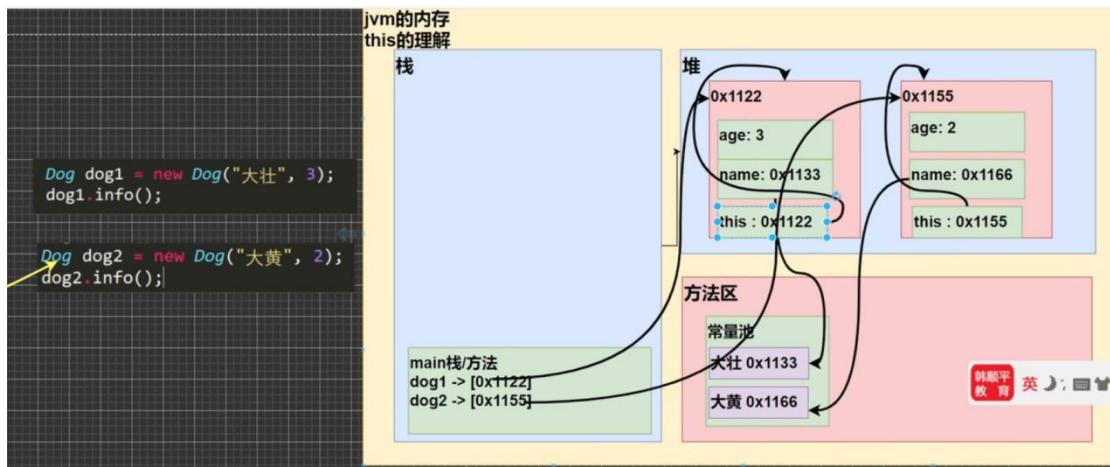
全局变量有默认值，局部变量没有，必须赋值。

构造器：

构造器是完成对象的初始化而不是创建对象。先在堆中创建对象，创建的对象中的属性赋初值，然后再将类中显式指定值的属性的值修改(显式初始化)，然后再通过构造方法给属性赋值，最后将对象的地址赋给栈帧中的对象引用。

一旦定义了自己的构造器，无参构造器就会被覆盖。

this 的理解：



this 是创建的对象中的一个隐含的属性。

this 关键字可以访问本类的构造器，但只能在构造方法中使用。this(参数列表)。

访问修饰符：

访问修饰符

- 4种访问修饰符的访问范围**

1	访问级别	访问控制修饰符	同类	同包	子类	不同包
2	公开	public	✓	✓	✓	✓
3	受保护	protected	✓	✓	✓	✗
4	默认	没有修饰符	✓	✓	✗	✗
5	私有	private	✓	✗	✗	✗

背下来!!!

- 使用的注意事项**

- 1) 修饰符可以用来修饰类中的属性，成员方法以及类
- 2) 只有默认的和public才能修饰类！，并且遵循上述访问权限的特点。
- 3) 因为没有学习继承，因此关于在子类中的访问权限，我们讲完子类后，在回头讲解
- 4) 成员方法的访问规则和属性完全一样。

继承：

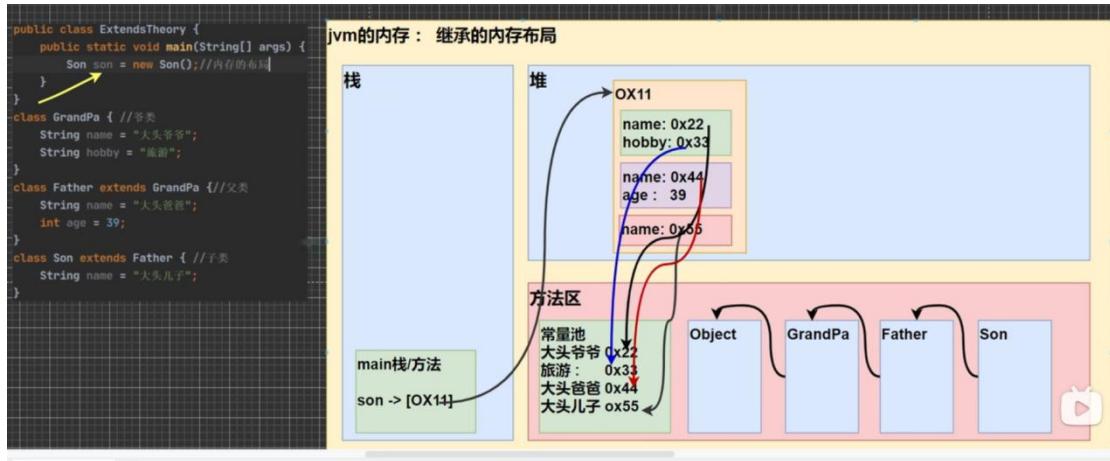
- 继承的深入讨论/细节问题**

2. 子类必须调用父类的构造器，完成父类的初始化
3. 当创建子类对象时，不管使用子类的那个构造器，默认情况下总会去调用父类的无参构造器，如果父类没有提供无参构造器，则必须在子类的构造器中用 super 去指定使用父类的哪个构造器完成对父类的初始化工作，否则，编译不会通过 [举例说明]

5. super在使用时，必须放在构造器第一行

6. super() 和 this() 都只能放在构造器第一行，因此这两个方法不能共存在一个构造器

因为 this()这个构造器中已经包含了 super()。



```
Son son = new Son(); //内存的布局
//?-> 这时请大家注意，要按照查找关系来返回信息
//(1) 首先看子类是否有该属性
//(2) 如果子类有这个属性，并且可以访问，则返回信息
//(3) 如果子类没有这个属性，就看父类有没有这个属性(如果父类有该属性，并且可以访问，就返回信息...)
//(4) 如果父类没有就按照(3)的规则，继续找上级父类，直到Object...
System.out.println(son.name); //返回就是大头儿子
```

重写：

● 注意事项和使用细节

方法重写也叫方法覆盖，需要满足下面的条件 //OverrideDetail.java

1. 子类的方法的参数,方法名称,要和父类方法的参数,方法名称完全一样。【演示】
2. 子类方法的返回类型和父类方法返回类型一样，或者是父类返回类型的子类
比如 父类 返回类型是 `Object` ,子类方法返回类型是 `String` 【演示】

```
public Object getInfo() {    public String getInfo(){
```

3. 子类方法不能缩小父类方法的访问权限 【演示】

```
void sayOk() {    public void sayOk(){
```

方法传参一般都是运用了多态，方法的参数如果是父类型，而实参却是子类型，如果在方法中父类调用了某个方法，而子类中重写的这个方法是私有的，则会造成错误。

如果子类不能访问父类中的某个方法，则不能重写这个方法。比如私有方法和不在同一包下的默认权限的方法。

构造方法也有 4 个权限修饰符，如果父类的构造方法都是默认或私有，则子类构造方法会报错，因为子类构造方法默认调用了父类的无参构造，但父类的构造器子类不能访问，所以父类的构造方法要设置为 `public` 或 `protected`。

子类可以继承父类的静态方法和属性，但不能重写静态方法。如果子类有和父类一样的静态方法和属性，则会覆盖掉父类的方法和属性。

名称	发生范围	方法名	形参列表	返回类型	修饰符
重载(overload)	本类	必须一样	类型,个数或者顺序至少有一个不同	无要求	无要求
重写(override)	父子类	必须一样	相同	子类重写的方法,返回的类型和父类返回的类型一致,或者是其子类	子类方法不能缩小父类方法的访问范围.

多态：

方法的多态：重写和重载（多个重载的方法，方法名相同，这就是多态。具有继承关系的类调用重写的方法，方法名相同，这也是多态）。

对象的多态：

面向对象编程-多态

- 多态的具体体现

2. 对象的多态 (核心, 困难, 重点)

老韩重要的几句话(记住):

- (1) 一个对象的编译类型和运行类型可以不一致
- (2) 编译类型在定义对象时, 就确定了, 不能改变
- (3) 运行类型是可以变化的.
- (4) 编译类型看定义时 = 号的左边, 运行类型看 = 号的右边

案例: `com.hspedu.poly_objpoly_ : PolyObject.java`

```
Animal animal = new Dog(); //编译时animal是Animal, 运行是animal是Dog
animal.cry();
//animal变成了Cat
animal = new Cat();
animal.cry();
```

动态绑定：（看视频）

java的动态绑定机制

1. 当调用对象方法的时候, 该方法会和该对象的内存地址/运行类型绑定
2. 当调用对象属性时, 没有动态绑定机制, 哪里声明, 那里使用

静态变量：

Java8 以前在方法区中, 以后在堆中 (class 对象中, 在类加载的时候就生成了)。

6. 类变量是在类加载时就初始化了, 也就是说, 即使你没有创建对象, 只要类加载了, 就可以使用类变量了。【案例演示】
7. 类变量的生命周期是随类的加载开始, 随着类消亡而销毁。

静态方法：

- 1) 类方法和普通方法都是随着类的加载而加载, 将结构信息存储在方法区：
类方法中无this的参数
普通方法中隐含着this的参数

代码块：

- (1) 下面的三个构造器都有相同的语句
- (2) 这样代码看起来比较冗余
- (3) 这时我们可以把相同的语句，放入到一个代码块中，即可
- (4) 这样当我们不管调用哪个构造器，创建对象，都会先调用代码块的内容
- (5) 代码块调用的顺序优先于构造器...

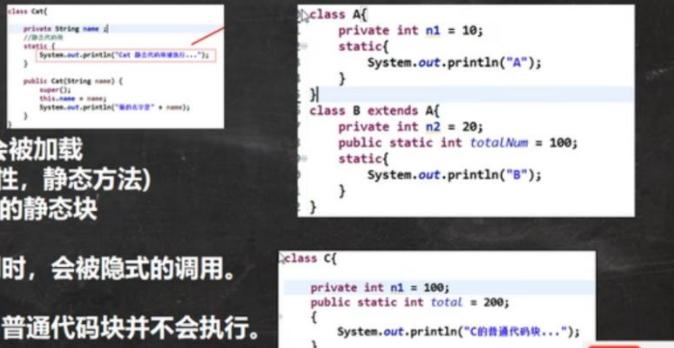
● 代码块使用注意事项和细节讨论 [CodeBlockDetail01.java](#) 烦人

- 1) static代码块也叫静态代码块，作用就是对类进行初始化，而且它随着类的加载而执行，并且只会执行一次，如果是普通代码块，每创建一个对象，就执行。

2) 类什么时候被加载

- ① 创建对象实例时(new)
- ② 创建子类对象实例，父类也会被加载
- ③ 使用类的静态成员时(静态属性，静态方法)

案例演示：A类 extends B类 的静态块



- 3) 普通的代码块，在创建对象实例时，会被隐式的调用。

被创建一次，就会调用一次。

如果只是使用类的静态成员时，普通代码块并不会执行。

4) 创建一个对象时，在一个类 调用顺序是:(重点，难点)：

- ① 调用静态代码块和静态属性初始化(注意：静态代码块和静态属性初始化调用的优先级一样，如果有多个静态代码块和多个静态变量初始化，则按他们定义的顺序调用)

- ② 调用普通代码块和普通属性的初始化(注意：普通代码块和普通属性初始化调用的优先级一样，如果有多个普通代码块和多个普通属性初始化，则按定义顺序调用)

- ③ 调用构造方法。

新写一个类演示 [【CodeBlockDetail02.java】](#)

- 5) 构造器的最前面其实隐含了 super() 和 调用普通代码块，新写一个类演示【截图+说明】，静态相关的代码块，属性初始化，在类加载时，就执行完毕，因此是优先于 构造器和普通代码块执行的 [【CodeBlockDetail03.java】](#)

```
class A {  
    public A() { //构造器  
        //这里有隐藏的执行要求  
        //((1) super(); //这个知识点，在前面讲解继承的时候，老师说  
        //((2) 调用普通代码块的 还有普通属性的初始化，按顺序  
        System.out.println("ok");  
    }  
}
```

6) 我们看一下创建一个子类时(继承关系), 他们的静态代码块, 静态属性初始化,

普通代码块, 普通属性初始化, 构造方法的调用顺序如下:

- ① 父类的静态代码块和静态属性(优先级一样, 按定义顺序执行)
- ② 子类的静态代码块和静态属性(优先级一样, 按定义顺序执行)
- ③ 父类的普通代码块和普通属性初始化(优先级一样, 按定义顺序执行)
- ④ 父类的构造方法
- ⑤ 子类的普通代码块和普通属性初始化(优先级一样, 按定义顺序执行)
- ⑥ 子类的构造方法 // 面试题

A, B, C 类 演示 [10Min]55 CodeBlockDetail04.java



测试代码.zip

7) 静态代码块只能直接调用静态成员(静态属性和静态方法), 普通代码块可以调用任意成员。

单例模式:

```
//步骤[单例模式-饿汉式]  
//1. 将构造器私有化  
//2. 在类的内部直接创建对象(该对象是static)  
//3. 提供一个公共的static方法, 返回 gf对象
```

饿汉式: 还不想使用对象, 对象就已经创建好了。

懒汉式: 使用时再创建对象。

通常对象都是重量级资源, 饿汉式可能造成资源浪费。单例模式可以节约资源。

● 饿汉式VS懒汉式

1. 二者最主要的区别在于创建对象的时机不同: 饿汉式是在类加载就创建了对象实例, 而懒汉式是在使用时才创建。
2. 饿汉式不存在线程安全问题, 懒汉式存在线程安全问题。(后面学习线程后, 会完善一把)
3. 饿汉式存在浪费资源的可能。因为如果程序员一个对象实例都没有使用, 那么饿汉式创建的对象就浪费了, 懒汉式是使用时才创建, 就不存在这个问题。
4. 在我们javaSE标准类中, java.lang.Runtime就是经典的单例模式。

```
if(instance == null){  
    synchronized (Bank.class) {  
        if(instance == null){  
            instance = new Bank();  
        }  
    }  
    return instance;
```

final 关键字：

在某些情况下，程序员可能有以下需求，就会使用到final：

- 1) 当不希望类被继承时，可以用final修饰。【案例演示】
- 2) 当不希望父类的某个方法被子类覆盖/重写(override)时，可以用final关键字修饰。【案例演示：访问修饰符 final 返回类型 方法名】
- 3) 当不希望类的某个属性的值被修改，可以用final修饰。【案例演示：public final double TAX_RATE=0.08】
- 4) 当不希望某个局部变量被修改，可以使用final修饰 【案例演示：final double TAX RATE=0.08】

1) final修饰的属性又叫常量，一般用 xx_xx_xx 来命名

2) final修饰的属性在定义时，必须赋初值，并且以后不能再修改，赋值可以在如下位置之一【选择一个位置赋初值即可】：

- ① 定义时：如 public final double TAX_RATE=0.08;
- ② 在构造器中
- ③ 在代码块中。

3) 如果final修饰的属性是静态的，则初始化的位置只能是
① 定义时 ② 在静态代码块 不能在构造器中赋值。

4) final类不能继承，但是可以实例化对象。[A2类]

5) 如果类不是final类，但是含有final方法，则该方法虽然不能重写，但是可以被继承。[A3类]

6) 一般来说，如果一个类已经是final类了，就没有必要再将方法修饰成final方法。

7) final 和 static 往往搭配使用，效率更高，不会导致类加载。底层编译器做了优化处理。

```
class Demo{  
    public static final int i=16; //  
    static{  
        System.out.println("韩顺平教育~");  
    }  
}
```

8) 包装类(Integer, Double, Float, Boolean 等都是final), String也是final类。

抽象类：

7) 如果一个类继承了抽象类，则它必须实现抽象类的所有抽象方法，除非它自己也声明为 abstract类。【举例 A类, B类, C类】

接口：

小结：

1. 在Jdk7.0前 接口里的所有方法都没有方法体。
2. Jdk8.0后接口类可以有静态方法，默认方法，也就是说接口中可以有方法的具体实现

默认方法和静态方法可以被继承。

```
interface MyInterface01 {  
    default public void myMethod(){  
        System.out.println("myInterface2");  
    }  
    public static void t2() {  
        System.out.println("t2~~~");  
    }  
}  
class A implements MyInterface01 {  
}
```

4) 抽象类实现接口，可以不用实现接口的方法。

6) 接口中的属性只能是final的，而且是 public static final 修饰符。比如：
int a=1; 实际上是 public static final int a=1; (必须初始化)

7) 接口中属性的访问形式：接口名.属性名

- 接口和继承解决的问题不同

继承的价值主要在于：解决代码的复用性和可维护性。

接口的价值主要在于：设计，设计好各种规范(方法)，让其它类去实现这些方法。即更加的灵活..

•

- 接口比继承更加灵活

接口比继承更加灵活，继承是满足 is - a 的关系，而接口只需满足 like - a 的关系。

- 接口在一定程度上实现代码解耦 [即接口规范性+动态绑定]



内部类：

类的五大成员：属性、方法、构造器、代码块、内部类。

● 内部类的分类

➤ 定义在外部类局部位置上 (比如方法内) :

- 1) 局部内部类 (有类名)
- 2) 匿名内部类 (没有类名, 重点!!!!!!)

➤ 定义在外部类的成员位置上:

- 1) 成员内部类 (没用static修饰)
- 2) 静态内部类 (使用static修饰)

局部内部类:

6. 外部其他类---不能访问---->局部内部类 (因为 局部内部类地位是一个局部变量)

7. 如果外部类和局部内部类的成员重名时, 默认遵循就近原则, 如果想访问外部类的成员, 则可以使用 (外部类名.this.成员) 去访问 【演示】
因为在内部类中, 所以this.n2是
是内部类的n2, 外部类名.this是
System.out.println("外部类的n2= " + 外部类名.this.n2); 外部对象(调用这个方法的对象)

● 局部内部类的使用 LocalInnerClass.java

说明: 局部内部类是定义在外部类的局部位置, 比如方法中, 并且有类名。

- 1. 可以直接访问外部类的所有成员, 包含私有的
- 2. 不能添加访问修饰符, 因为它的地位就是一个局部变量。局部变量是不能使用修饰符的。但是可以使用final修饰, 因为局部变量也可以使用final
- 3. 作用域: 仅仅在定义它的方法或代码块中。

4. 局部内部类---访问---->外部类的成员 [访问方式: 直接访问]

5. 外部类---访问---->局部内部类的成员

访问方式: 创建对象, 再访问(注意: 必须在作用域内)

```
class Outer03 { // 外部类
    private int n1 = 10;
    private static String name = "张三";
    public void say() {
        int n3 = 30;
        class LocalInner01 { //相当于一个局部变量
            int n2 = 40;
            public void show() {
                System.out.print("张三");
            }
        }
        LocalInner01 localInner01 = new LocalInner01();
        localInner01.show();
    }
}
```

匿名内部类:

```
//7. jdk底层在创建匿名内部类 Outer04$1, 立即马上就创建了 Outer04$1实例, 并且把地址
//    返回给 tiger
//8. 匿名内部类使用一次, 就不能再使用
IA tiger = new IA() {
    @Override
    public void cry() {
        System.out.println("老虎叫唤...");
```

匿名内部类不仅可以通过实现接口来实现，还可以继承类来实现。

- 3. 可以直接访问外部类的所有成员，包含私有的 [案例演示]
 - 4. 不能添加访问修饰符,因为它的地位就是一个局部变量。 [过]
 - 5. 作用域 : 仅仅在定义它的方法或代码块中。 [过]
 - 6. 匿名内部类---访问---->外部类成员 [访问方式: 直接访问]
 - 7. 外部其他类---不能访问---->匿名内部类 (因为 匿名内部类地位是一个局部变量)
 - 8. 如果外部类和匿名内部类的成员重名时, 匿名内部类访问的话, 默认遵循就近原则, 如果想访问外部类的成员, 则可以使用 (外部类名.this.成员) 去访问
- 成员内部类：
- 说明：成员内部类是定义在外部类的成员位置，并且没有static修饰。
1. 可以直接访问外部类的所有成员，包含私有的
- ```
class Outer01{ //外部类
private int n1 = 10;
public String name = "张三";
class Innter01{
 public void say(){
 System.out.println("Outer01 的 n1 = " + n1 + " outer01 的 name = " + name);
 }
}
```
2. 可以添加任意访问修饰符(public、protected、默认、private),因为它的地位就是一个成员。

## 内部类

- 3. 作用域 MemberInnerClass01.java  
和外部类的其他成员一样，为整个类体  
比如前面案例，在外部类的成员方法中创建成员内部类对象，再调用方法。

- 4. 成员内部类---访问---->外部类成员(比如：  
属性) [访问方式：直接访问] (说明)

- 5. 外部类---访问----->成员内部类 (说明)  
访问方式：创建对象，再访问

- 6. 外部其他类---访问---->成员内部类

```
public class TestInner3 {
 public static void main(String[] args) {

 //其它外部类访问内部类，方式1
 Outer01 outer01 = new Outer01();
 Innter01 innter01 = outer01.new Innter01();

 //方式2：
 Innter01 innter012 = new Outer01().new Innter01();

 //方式3：使用一个方法来获取，更加简洁
 Innter01 innter01Instance = new Outer01().getInnter01Instance();

 }
}
class Outer01{
 private int n1 = 10;
 public String name = "张三";
}
class Innter01{
 public void say(){
 System.out.println("Outer01 的 n1 = " + n1 +
 " outer01 的 name = " + name);
 }
 public Innter01 getInnter01Instance(){
 return new Innter01();
 }
 private void show(){
 Innter01 innter01 = new Innter01();
 innter01.say();
 }
}
```

```
// outer08.new Inner08(); 相当于把 new Inner08()当做是outer08成员
// 这就是一个语法，不要特别的纠结。
Outer08.Inner08 inner08 = outer08.new Inner08();
```

- 7. 如果外部类和内部类的成员重名时，内部类访问的话，默认遵循就近原则，如。  
如果想访问外部类的成员，则可以使用 (外部类名.this.成员) 去访问

静态内部类：

### ● 静态内部类的使用 StaticInnerClass01.java

说明：静态内部类是定义在外部类的成员位置，并且有static修饰

- 1. 可以直接访问外部类的所有静态成员，包含私有的，但不能直接访问非静态成员
- 2. 可以添加任意访问修饰符(public、protected、默认、private)，因为它的地位就是一个成员。
- 3. 作用域：同其他的成员，为整个类体

```
class Outer02{ //外部类
 private int n1 = 10;
 private static String name = "张三";
 static class Inner02{
 public void say(){
 System.out.println(name);
 //不能直接访问外部类的非静态成员
 //System.out.println(n1);
 }
 public void show(){
 //外部类使用内部类
 new Inner02().say();
 }
 }
}
```

## 6. 外部其他类---访问---->静态内部类

```
public class TestInnerDemo3 {
 public static void main(String[] args) {

 //两种方式

 B02 b02_1 = new A03.B02(); //这里就不会创建一个 A03对象实例
 b02_1.say();

 B02 b02_2 = new A03().getB02Object(); //第二方式

 }
}
```

//方式 1，通过一个方法来获取一个静态内部类的对象实例  
Innter02 inner02 = Outer02.getInnter02();  
inner02.say();

- 7. 如果外部类和静态内部类的成员重名时，静态内部类访问的时，默认遵循就近原则，如果想访问外部类的成员，则可以使用（外部类名.成员）去访问

```
class Outer02{ //外部类
 static class Inner02{ //静态内部类
 void show() {
 System.out.println("Inner02");
 }
 }
}
```

因为只能访问静态成员

//注意没有生成Outer02类的对象实例

枚举：

## 自定义类实现枚举-应用案例

- 1. 不需要提供setXxx方法，因为枚举对象值通常为只读。
- 2. 对枚举对象/属性使用 final + static 共同修饰，实现底层优化。
- 3. 枚举对象名通常使用全部大写，常量的命名规范。
- 4. 枚举对象根据需要，也可以有多个属性 //Enumeration02.java

```
class Season{
 private String name;//季节名称
 private String description;//季节描述
 public String getName() {
 return name;
 }
 public String getDescription() {
 return description;
 }
 private Season(String name, String description){
 this.name=name;
 this.description=description;
 }
 public final static Season SPRING = new Season("春天", "温暖");
 public final static Season SUMMER = new Season("夏天", "炎热");
 public final static Season AUTUMN = new Season("秋天", "凉爽");
 public final static Season WINTER = new Season("冬天", "寒冷");
}
```

## enum关键字实现枚举-快速入门

使用enum 来实现前面的枚举案例，看老师演示，主要体会和自定义类实现枚举不同的地方。 Enumeration03.java

```
enum Season2 {
 // public final static Season2 SPRING = new Season2("春天", "温暖");
 // 因为使用了enum ,因此上面的代码可以简化成如下形式
 SPRING("春天", "温暖"),
 SUMMER("夏天", "炎热"),
 AUTUMN("秋天", "凉爽"),
 WINTER("冬天", "寒冷");
 private Season2(String name, String description) {
 this.name = name;
 this.description = description;
 }
 private String name;// 季节名称
 private String description;// 季节描述
 public String getName() {
 return name;
 }
 public String getDescription() {
 return description;
 }
}
```

```
//如果使用了enum 来实现枚举类
//1. 使用关键字 enum 替代 class
//2. public static final Season SPRING = new Season("春天", "温暖") 直接使用
// SPRING("春天", "温暖") 解读 常量名(实参列表)
//3. 如果有多个常量(对象), 使用 ,号间隔即可
//4. 如果使用enum 来实现枚举, 要求将定义常量对象, 写在前面
SPRING("春天", "温暖"), WINTER("冬天", "寒冷"), AUTUMN("秋天", "凉爽"), SUMMER("夏天", "炎热");
private String name;
private String desc;//描述
```

## ● enum关键字实现枚举注意事项

1. 当我们使用enum 关键字开发一个枚举类时, 默认会继承Enum类[如何证明]
2. 传统的 public static final Season2 SPRING = new Season2("春天", "温暖"); 简化成 SPRING("春天", "温暖"), 这里必须知道, 它调用的是哪个构造器.
3. 如果使用无参构造器 创建 枚举对象, 则实参列表和小括号都可以省略
4. 当有多个枚举对象时, 使用,间隔, 最后有一个分号结尾
5. 枚举对象必须放在枚举类的行首.

注解:

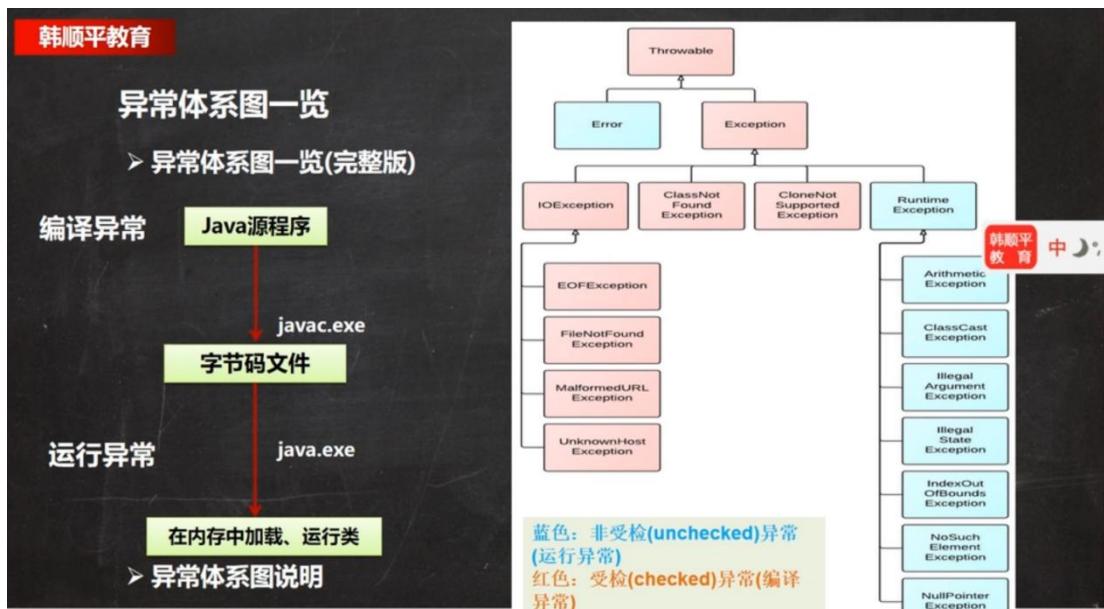
### ➤ Override 使用说明

1. @Override 表示指定重写父类的方法 (从编译层面验证), 如果父类没有fly方法, 则会报错
2. 如果不写@Override 注解, 而父类仍有 public void fly(){}, 仍然构成重写
3. @Override 只能修饰方法, 不能修饰其它类, 包, 属性等等
4. 查看@Override注解源码为 @Target(ElementType.METHOD),说明只能修饰方法
5. @Target 是修饰注解的注解, 称为元注解

## ● 元注解的种类 (使用不多, 了解, 不用深入研究)

- 1) Retention //指定注解的作用范围, 三种 SOURCE,CLASS,RUNTIME
- 2) Target // 指定注解可以在哪些地方使用
- 3) Documented //指定该注解是否会在javadoc体现
- 4) Inherited //子类会继承父类注解

异常：



## ● 常见的运行时异常包括

- 1) **NullPointerException** 空指针异常
- 2) **ArithmetricException** 数学运算异常
- 3) **ArrayIndexOutOfBoundsException** 数组下标越界异常
- 4) **ClassCastException** 类型转换异常
- 5) **NumberFormatException** 数字格式不正确异常[]

## ● 常见的编译异常

编译异常是由于底层源码向上抛出异常导致的  
可以try-catch或者向上throws

- ✓ **SQLException** //操作数据库时，查询表可能发生异常
- ✓ **IOException** //操作文件时，发生的异常
- ✓ **FileNotFoundException** //当操作一个不存在的文件时，发生异常
- ✓ **ClassNotFoundException** //加载类，而该类不存在时，异常
- ✓ **EOFException** //操作文件，到文件末尾，发生异常
- ✓ **IllegalArgumentException** //参数异常

### ● try-catch方式处理异常-注意事项 TryCatchDetail

- 1) 如果异常发生了，则异常发生后面的代码不会执行，直接进入到catch块。
- 2) 如果异常没有发生，则顺序执行try的代码块，不会进入到catch。
- 3) 如果希望不管是否发生异常，都执行某段代码(比如关闭连接，释放资源等)  
则使用如下代码- finally { }

- 5) 可以进行 try-finally 配合使用，这种用法相当于没有捕获异常，因此程序会直接崩掉。应用场景，就是执行一段代码，不管是否发生异常，都必须执行某个业务逻辑

```

public class ExceptionExe01 {
 public static int method() {
 int i = 1; // i = 1
 try {
 i++; // i = 2
 String[] names = new String[3];
 if (names[1].equals("tom")) { // 空指针
 System.out.println(names[1]);
 } else {
 names[3] = "hspedu";
 }
 return 1;
 } catch (ArrayIndexOutOfBoundsException e) {
 return 2;
 } catch (NullPointerException e) {
 return ++i; // i = 3 => 保存临时变量 temp = 3;
 } finally {
 ++i; // i = 4
 System.out.println("i=" + i); // i = 4
 }
 }
 public static void main(String[] args) {
 System.out.println(method()); // 3
 }
} // 练习3 i = 4, 3

```

### ● 注意事项和使用细节 **ThrowsDetail.java**

- 1) 对于编译异常，程序中必须处理，比如 try-catch 或者 throws
- 2) 对于运行时异常，程序中如果没有处理，默认就是 throws 的方式处理 [举例]
- 3) 子类重写父类的方法时，对抛出异常的规定：子类重写的方法，所抛出的异常类型要么和父类抛出的异常一致，要么为父类抛出的异常类型的子类型 [举例]
- 4) 在 throws 过程中，如果有方法 try-catch，就相当于处理异常，就可以不必 throws

```

public void test() /*throws ArithmeticException*/ {
 int n1 = 9 / 0; // 发生运行异常，你没有 catch 默认就是 throws
}

```

```

class Father {
 public void method() throws RuntimeException {}
}
class Son extends Father {
 public void method() throws NullPointerException {}
}

```

### ● 自定义异常的步骤

- 1) 定义类：自定义异常类名（程序员自己写）继承 Exception 或 RuntimeException
- 2) 如果继承 Exception，属于编译异常
- 3) 如果继承 RuntimeException，属于运行异常（一般来说，继承 RuntimeException）

包装类：

韩顺平教育

## 包装类

- 包装类的分类 **WrapperType.java**
- 1. 针对八种基本数据类型相应的引用类型—包装类
- 2. 有了类的特点，就可以调用类中的方法。

| 基本数据类型  | 包装类       |
|---------|-----------|
| boolean | Boolean   |
| char    | Character |
| byte    | Byte      |
| short   | Short     |
| int     | Integer   |
| long    | Long      |
| float   | Float     |
| double  | Double    |

父类:Number

如下两个题目输出结果相同吗？各是什么？

Object obj1 = true? new Integer(1) : new Double(2.0); //三元运算符【是一个整体】  
大师  
System.out.println(obj1); // 什么？1.0

```
Object obj2;
if(true)
 obj2 = new Integer(1);
else
 obj2 = new Double(2.0);
System.out.println(obj2); //1
输出什么？1， 分别计算
```

包装类：

看看下面代码，输出什么结果？为什么？2min

```
public void method1() {
 Integer i = new Integer(1);
 Integer j = new Integer(1);
 System.out.println(i == j); //False
 //所以，这里主要是看范围 -128 ~ 127 就是直接返回
 Integer m = 1; //底层 Integer.valueOf(1); -> 阅读源码
 Integer n = 1;//底层 Integer.valueOf(1);
 System.out.println(m == n); //T
 //所以，这里主要是看范围 -128 ~ 127 就是直接返回
 //，否则，就new Integer(xx);
 Integer x = 128;//底层Integer.valueOf(1);
 Integer y = 128;//底层Integer.valueOf(1);
 System.out.println(x == y); //False
}
```

如果有基本数据类型，则直接判断值的大小是否相等。

String:

```
//6. String 是final 类, 不能被其他的类继承
//7. String 有属性 private final char value[]; 用于存放字符串内容
//8. 一定要注意: value 是一个final类型, 不可以修改(需要功力): 即value不能指向
// 新的地址, 但是单个字符内容是可以变化
```

final 修饰的对象地址不能改变, 但内容可以改变, 比如 final 修饰的数组, 数组的内容可以改变, 但不能指向其他的数组。

## ● 两种创建String对象的区别

方式一: 直接赋值 String s = "hsp";

方式二: 调用构造器 String s2 = new String("hsp");

1. 方式一: 先从常量池查看是否有"hsp" 数据空间, 如果有, 直接指向; 如果没有则重新创建, 然后指向。s最终指向的是常量池的空间地址
2. 方式二: 先在堆中创建空间, 里面维护了value属性, 指向常量池的hsp空间。如果常量池没有"hsp", 重新创建, 如果有, 直接通过value指向。最终指向的是堆中的空间地址。

方式一: String a = "aaa";

方式二: String b = new String("aaa");

两种方式都能创建字符串对象, 但方式一要比方式二更优。

因为字符串是保存在常量池中的, 而通过new创建的对象会存放在堆内存中。

常量池中已经有字符串常量"aaa"

通过方式一创建对象, 程序运行时会在常量池中查找"aaa"字符串, 将找到的"aaa"字符串的地址赋给a。

通过方式二创建对象, 无论常量池中有没有"aaa"字符串, 程序都会在堆内存中开辟一片新空间存放新对象。

常量池中没有字符串常量"aaa"

通过方式一创建对象, 程序运行时会将"aaa"字符串放进常量池, 再将其地址赋给a。

通过方式二创建对象, 程序会在堆内存中开辟一片新空间存放新对象, 同时会将"aaa"字符串放入常量池, 相当于创建了两个对象。

String有两种赋值方式

1.直接赋值: String str = "hello world";

2.构造方法赋值: String str = new String("hello world");

字符串常量实际上就是String的匿名对象, 因为字符串常量可以直接调用方法, 常见的为"a".equals(Object)。

使用直接赋值的方式创建对象, 除了开辟堆空间外, 还会实现堆空间的重用。这是因为在JVM的底层会存在一个对象池(不一定只保存String对象), 当代码中使用了直接赋值的方式定义了一个String对象, 系统会将这个匿名对象放入对象池保存, 如果后续还有其他String类型对象也采用了直接赋值且内容一致时, 将不会开辟新的堆空间而是将已有的对象进行引用的分配, 从而继续使用。

使用构造方法创建对象时, 和普通对象的创建过程没有任何区别, 不会将这个字符串放入对象池, 但是我们可以手动入池, 即使用String中的intern()方法将字符串对象放进对象池中供后续使用

常量池中的字符串也是对象, 堆中创建的对象的 value 数组和常量池中的字符串对象的 value 数组指向是一样的。

```
public String(@NotNull String original) {
 this.value = original.value;
 this.hash = original.hash;
}
```

上面三张图都有小瑕疵，但结合起来是对的。

当调用 intern 方法时，如果池已经包含一个等于此 String 对象的字符串（用 equals(Object) 方法确定），则返回池中的字符串。否则，将此 String 对象添加到池中，并返回此 String 对象的引用  
解读：(1) b.intern() 方法最终返回的是常量池的地址（对象）。

```
String a = "hello"+ "abc";
创建了几个对象？只有1个对象。1min
•//老韩解读：String a = "hello"+ "abc"; //==>优化等价 String a = "helloabc";
```

分析

1. 编译器不傻，做一个优化，判断创建的常量池对象，是否有引用指向
2. String a = "hello"+ "abc"; => String a = "helloabc";

变量+变量 或者 变量+常量：

先创建一个空的 StringBuilder，然后调用两次 append，最后调用 sb 的 toString 方法返回

```
public String toString() {
 // Create a copy, don't share the array
 return new String(value, offset: 0, count);
}
```

intern 说明：

1.6 及之前：常量池中有，则返回常量池中的引用，如果没有，则在常量池中创建并返回。  
1.7 及之后：常量池中有，则返回常量池中的引用，如果没有，则常量池中不创建对象，而是创建一个引用，指向调用 intern 的那个堆中的字符串对象。但如果在 intern 之前出现过“字符串”字面量，则会在常量池中创建。

StringBuffer：

## ● String VS StringBuffer

- 1) String保存的是字符串常量，里面的值不能更改，每次String类的更新实际上就是更改地址，效率较低 //private final char value[];
- 2) StringBuffer保存的是字符串变量，里面的值可以更改，每次StringBuffer的更新实际上可以更新内容，不用每次更新地址，效率较高 //char[] value; //这个放在堆。
- String修改时直接重新new一个对象，而StringBuilder是修改value数组的指向

## ● String、StringBuffer 和StringBuilder的比较

- 1) StringBuilder 和 StringBuffer 非常类似，均代表可变的字符序列，而且方法也一样
- 2) String: 不可变字符序列，效率低，但是复用率高。
- 3) StringBuffer: 可变字符序列、效率较高(增删)、线程安全，看源码
- 4) StringBuilder: 可变字符序列、效率最高、线程不安全
- 5) String使用注意说明：

```
string s="a"; //创建了一个字符串
s += "b"; //实际上原来的"a"字符串对象已经丢弃了，现在又产生了一个字符串s+"b" (也就是"ab")。如果多次执行这些改变串内容的操作，会导致大量副本字符串对象存留在内存中，降低效率。如果这样的操作放到循环中，会极大影响程序的性能 => 结论：如果我们对String 做大量修改，不要使用String
```

大数据处理方法：

## • 应用场景：

- 1) BigInteger适合保存比较大的整型
- 2) BigDecimal适合保存精度更高的浮点型（小数）

BigDecimal 除法除不尽会报错，可以选择有两个参数得除法函数来指定保留几位小数。

日期类：

## ● 第一代日期类

- 1) Date: 精确到毫秒，代表特定的瞬间
- 2) SimpleDateFormat: 格式和解析日期的类  
SimpleDateFormat 格式化和解析日期的具体类。它允许进行格式化（日期 -> 文本）、解析（文本 -> 日期）和规范化。

| 字母 | 日期或时间元素            | 表示                       | 示例                                    |
|----|--------------------|--------------------------|---------------------------------------|
| G  | Era 标志符            | <u>Text</u>              | AD                                    |
| y  | 年                  | <u>Year</u>              | 1996; 96                              |
| M  | 年中的月份              | <u>Month</u>             | July; Jul; 07                         |
| w  | 年中的周数              | <u>Number</u>            | 27                                    |
| W  | 月份中的周数             | <u>Number</u>            | 2                                     |
| D  | 年中的天数              | <u>Number</u>            | 189                                   |
| d  | 月份中的天数             | <u>Number</u>            | 10                                    |
| F  | 月份中的星期             | <u>Number</u>            | 2                                     |
| E  | 星期中的天数             | <u>Text</u>              | Tuesday; Tue                          |
| a  | Am/pm 标记           | <u>Text</u>              | PM                                    |
| H  | 一天中的小时数 (0-23)     | <u>Number</u>            | 0                                     |
| k  | 一天中的小时数 (1-24)     | <u>Number</u>            | 24                                    |
| K  | am/pm 中的小时数 (0-11) | <u>Number</u>            | 0                                     |
| h  | am/pm 中的小时数 (1-12) | <u>Number</u>            | 12                                    |
| m  | 小时中的分钟数            | <u>Number</u>            | 30                                    |
| s  | 分钟中的秒数             | <u>Number</u>            | 55                                    |
| S  | 毫秒数                | <u>Number</u>            | 978                                   |
| z  | 时区                 | <u>General time zone</u> | Pacific Standard Time; PST; GMT-08:00 |
| Z  | 时区                 | <u>RFC 822 time zone</u> | -0800                                 |

```

Date d1 = new Date(); //获取当前系统时间
System.out.println("当前日期=" + d1);
Date d2 = new Date(9234567); //通过指定毫秒数得到时间
System.out.println(d1.getTime()); //获取某个时间对应的毫秒数

//老韩解读
//1. 创建 SimpleDateFormat对象，可以指定相应的格式
//2. 这里的格式使用的字母是规定好，不能乱写

SimpleDateFormat sdf = new SimpleDateFormat("yyyy年MM月dd日 hh:mm:ss E");
String format = sdf.format(d1); // format:将日期转换成指定格式的字符串
System.out.println("当前日期=" + format); Date转String

String s = "1996年01月01日 10:20:30 星期一"; String转Date
Date parse = sdf.parse(s);

```

### 3) 应用实例

```
Calendar c = Calendar.getInstance(); //创建日历类对象//比较简单，自由
System.out.println(c);
//2.获取日历对象的某个日历字段
System.out.println("年: "+c.get(Calendar.YEAR));
System.out.println("月: "+(c.get(Calendar.MONTH)+1));
System.out.println("日: "+c.get(Calendar.DAY_OF_MONTH));
System.out.println("小时: "+c.get(Calendar.HOUR));
System.out.println("分钟: "+c.get(Calendar.MINUTE));
System.out.println("秒: "+c.get(Calendar.SECOND));
//Calendar 没有专门的格式化方法，所以需要程序员自己来组合显示
System.out.println(c.get(Calendar.YEAR)+"年"+(c.get(Calendar.MONTH)+1)+"月"
"+c.get(Calendar.DAY_OF_MONTH)+"日");
```

- 第三代日期类

➤ 前面两代日期类的不足分析

JDK 1.0中包含了一个java.util.Date类，但是它的大多数方法已经在JDK 1.1引入  
Calendar类之后被弃用了。而Calendar也存在问题是：

- 1) 可变性：像日期和时间这样的类应该是不可变的。
- 2) 偏移性：Date中的年份是从1900开始的，而月份都从0开始。
- 3) 格式化：格式化只对Date有用，Calendar则不行。
- 4) 此外，它们也不是线程安全的；不能处理闰秒等（每隔2天，多出1s）。



➤ 第三代日期类常见方法

1) **LocalDate**(日期)、**LocalTime**(时间)、**LocalDateTime**(日期时间) JDK8加入

LocalDate只包含日期，可以获取日期字段

- LocalTime只包含时间，可以获取时间字段

LocalDateTime包含日期+时间，可以获取日期和时间字段

案例演示[后ppt]:

```
LocalDateTime ldt = LocalDateTime.now(); //LocalDate.now();//LocalTime.now()
System.out.println(ldt);
ldt.getYear();ldt.getMonthValue();ldt.getMonth();ldt.getDayOfMonth();
ldt.getHour();ldt.getMinute();ldt.getSecond();
```



2) **DateTimeFormatter**格式日期类

类似于**SimpleDateFormat**

```
DateTimeFormat dtf = DateTimeFormatter.ofPattern(格式);
String str = dtf.format(日期对象);
```

### 3) Instant 时间戳

I

类似于Date

提供了一系列和Date类转换的方式

Instant——>Date:

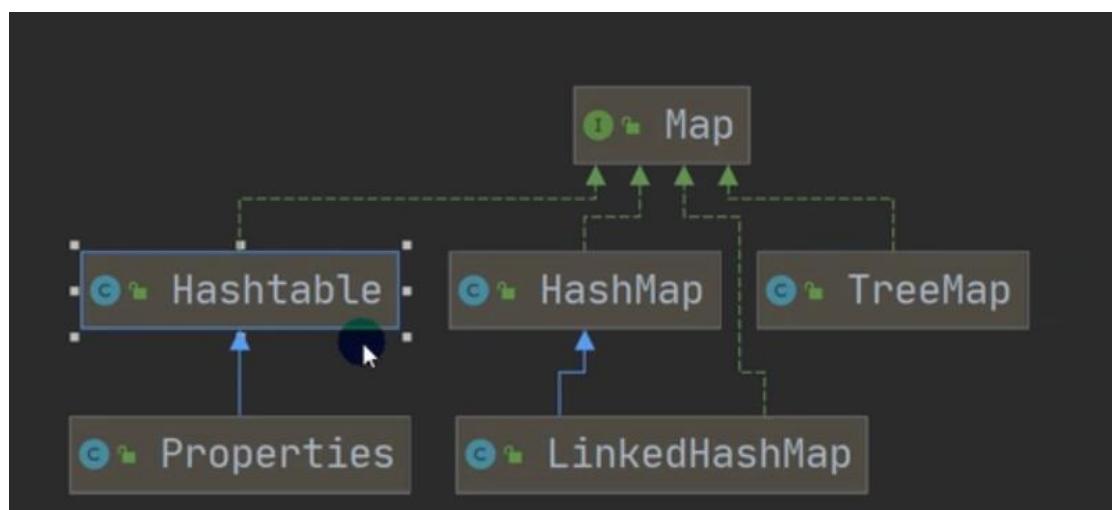
Date date = Date.from(instant);

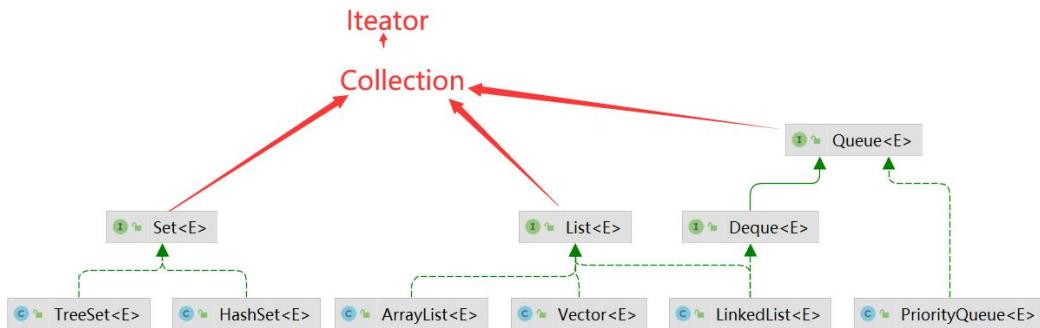
Date——>Instant:

Instant instant = date.toInstant();

```
//1. 通过 静态方法 now() 获取表示当前时间戳的对象
Instant now = Instant.now();
System.out.println(now);
//2. 通过 from 可以把 Instant转成 Date
Date date = Date.from(now);
//3. 通过 date的toInstant() 可以把 date 转成Instant对象
Instant instant = date.toInstant();
```

集合：





泛型：

```

//使用传统的方法来解决====> 使用泛型
//老韩解读
//1. 当我们 ArrayList<Dog> 表示存放到 ArrayList 集合中的元素是Dog类型
//2. 如果编译器发现添加的类型，不满足要求，就会报错
ArrayList<Dog> arrayList = new ArrayList<Dog>();
arrayList.add(new Dog("旺财", 10));
arrayList.add(new Dog("发财", 1));
arrayList.add(new Dog("小黄", 5));
//假如我们的程序员，不小心，添加了一只猫
arrayList.add(new Cat("招财猫", 8));

```

### ● 泛型的好处

- 1) 编译时，检查添加元素的类型，提高了安全性
- 2) 减少了类型转换的次数，提高效率 [说明]

✓ 不使用泛型  
 Dog -加入-> Object -取出-> Dog //放入到ArrayList 会先转成 Object，在取出时，还需要转换成Dog  
 ✓ 使用泛型  
 Dog -> Dog -> Dog // 放入时，和取出时，不需要类型转换，提高效率

### 3) 不再提示编译警告

```

//遍历
for (Object o : arrayList) {
 //向下转型Object ->Dog
 Dog dog = (Dog) o; 有了泛型不用向下转型
 System.out.println(dog.getName() + " - " + dog.getAge());
}

```

- 4) 泛型的作用是：可以在类声明时通过一个标识表示类中某个属性的类型，或者是某个方法的返回值的类型，或者是参数类型。[有点难，举例 Generic03.java]

1. interface List<T>{} , public class HashSet<E>{}.. 等等  
说明: T, E 只能是引用类型  
看看下面语句是否正确?:  
List<Integer> list = new ArrayList<Integer>();  
List<int> list2 = new ArrayList<int>();
2. 在指定泛型具体类型后, 可以传入该类型或者其子类类型
3. 泛型使用形式  
List<Integer> list1 = new ArrayList<Integer>();  
List<Integer> list2 = new ArrayList<>(); [说明:]
3. 如果我们这样写 List list3 = new ArrayList(); 默认给它的 泛型是 [<E> E就是 Object ]  
即:

#### ➤ 注意细节

- 1) 普通成员可以使用泛型 (属性、方法)
- 2) 使用泛型的数组, 不能初始化 泛型成员都不能初始化, 因为不知道要开辟多大空间
- 3) 静态方法中不能使用类的泛型 静态属性也不能用泛型
- 4) 泛型类的类型, 是在创建对象时确定的(因为创建对象时, 需要指定确定类型)
- 5) 如果在创建对象时, 没有指定类型, 默认为Object

```
interface IA extends IUsb<String, Double> {
}

//当我们去实现IA接口时, 因为IA在继承IUsb 接口时, 指定了U 为String R为Double
//, 在实现IUsb接口的方法时, 使用String替换U, 是Double替换R
class AA implements IA {

 @Override
 public Double get(String s) {
 return null;
 }
 @Override
 public void hi(Double aDouble) {
 }
 @Override
 public void run(Double r1, Double r2, String u1, String u2) {

```

韩顺平 教育 英

自定义泛型方法:

## ● 自定义泛型方法

### ➤ 基本语法

修饰符 <T,R..> 返回类型 方法名(参数列表) {  
}

### ➤ 注意细节

1. 泛型方法，可以定义在普通类中，也可以定义在泛型类中
2. 当泛型方法被调用时，类型会确定
3. public void eat(E e) {}, 修饰符后没有<T,R..> eat方法不是泛型方法，而是使用了泛型

### ➤ 应用案例 CustomMethodGeneric.java

看老师演示.

```
class Bird<T,R,M>{
 public<E> void fly(E t){
 System.out.println(t);
 }

 class Fish {
 public<A> A show(A t){
 System.out.println("t的值: "+t);
 System.out.println("t的类型: "
 +t.getClass().getSimpleName());
 return t;
 }
 }
}
```

```
class Fish<T, R> {//泛型类
 public void run() {//普通方法
 }
 public<U,M> void eat(U u, M m) {//泛型方法
 }
 //说明
 //1. 下面hi方法不是泛型方法
 //2. 是hi方法使用了类声明的 泛型
 public void hi(T t) {
 }
 //泛型方法，可以使用类声明的泛型，也可以使用自己声明泛型
 public<K> void hello(R r, K k) {
 }
}
```

1) 泛型不具备继承性

```
List<Object> list = new ArrayList<String>(); // 对吗?
```

2) <?> : 支持任意泛型类型

3) <? extends A>: 支持A类以及A类的子类，规定了泛型的上限

4) <? super A>: 支持A类以及A类的父类，不限于直接父类，规定了泛型的下限

### ● 应用案例

```
class AA {
}
class BB extends AA {
}
class CC extends BB {
}
```

```
public static void printCollection1(List<?> c) {
 for (Object object : c) { // 通配符，取出时，就是Object
 System.out.println(object);
 }
}
// ? extends AA 表示 上限，可以接受 AA或者AA子类
public static void printCollection2(List<? extends AA> c) {
 for (Object object : c) {
 System.out.println(object);
 }
}
// ? super 子类类名AA:支持AA类以及AA类的父类，不限于直接父类，
// 规定了泛型的下限
public static void printCollection3(List<? super AA> c) {
 for (Object object : c) {
 System.out.println(object);
 }
}
```

泛型的通配符和类型一起使用，用在方法的参数或返回类

型上



```
ArrayList<Object> a11 = new ArrayList<Object>();
ArrayList<String> a12 = new ArrayList<String>();
ArrayList<BB> a13 = new ArrayList<BB>();
ArrayList<BB> a14 = new ArrayList<BB>();
ArrayList<CC> a15 = new ArrayList<CC>();
```

线程基础：

## ● 创建线程的两种方式

在java中线程来使用有两种方法。

1. 继承Thread类，重写run方法

2. 实现Runnable接口，重写run方法

```
@Override
```

```
public void run() {
 if (target != null) {
 target.run();
 }
}
```

```
/* What will be run. */
```

```
private Runnable target;
```

```
public Thread(Runnable target) {
 init(g: null, target, name: "Thread")
}
```

继承 Thread 类，调用 start() 时内部调用 Thread 重写的 run()

实现 Runnable 接口，作为参数传到 Thread 的带参构造器中，调用 start() 时调用 target 的 run()，其中 Thread 带参构造器中的 target 会赋给 Thread 中的 target 属性。

### 3) 使用 JConsole 监控线程执行情况，并画出程序示意图！

主线程结束，如果有其他线程，则 java 进程不会结束。

(1)

```
public synchronized void start() {
 start0();
}
```

(2)

```
//start0() 是本地方法，是JVM调用，底层是c/c++实现
//真正实现多线程的效果，是start0()，而不是 start()
private native void start0();
```

start() 方法调用 start0() 方法后，该线程并不一定会立马执行，只是将线程变成了可运行状态。具体什么时候执行，取决于 CPU，由 CPU 统一调度。

1. java 是单继承的，在某些情况下一个类可能已经继承了某个父类，这时在用继承 Thread 类方法来创建线程显然不可能了。
2. java 设计者们提供了另外一个方式创建线程，就是通过实现 Runnable 接口来创建线程
2. 实现 Runnable 接口方式更加适合多个线程共享一个资源的情况，并且避免了单继承的限制

```
T3 t3 = new T3("hello ");
Thread thread01 = new Thread(t3);
Thread thread02 = new Thread(t3);
thread01.start();
thread02.start();
```

两个线程共享一个 t3 对象

```
try {
 System.out.println(Thread.currentThread().getName() + " 休眠中~~~");
 Thread.sleep(20000); //20秒 中断和终止不一样，中断是打断一个正在sleep的线程,
} catch (InterruptedException e) { sleep的线程会捕获一个异常
 //当该线程执行到一个interrupt 方法时，就会catch 一个 异常，可以加入自己的业务代码
 System.out.println(Thread.currentThread().getName() + "被 interrupt了");
}
```

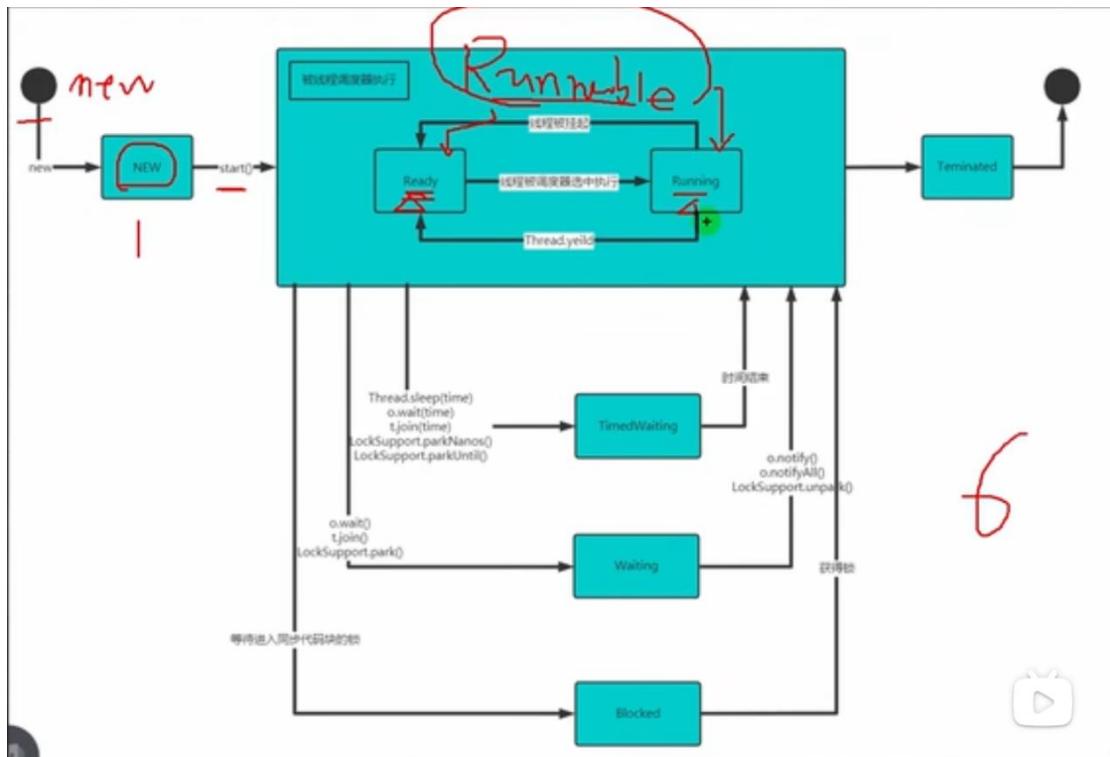
```
//如果我们希望当main线程结束后，子线程自动结束
//，只需将子线程设为守护线程即可
myDaemonThread.setDaemon(true);
myDaemonThread.start();
```

```
public static enum Thread.State
extends Enum<Thread.State>
```

线程状态。 线程可以处于以下状态之一：

- NEW  
尚未启动的线程处于此状态。
- RUNNABLE  
在Java虚拟机中执行的线程处于此状态。
- BLOCKED  
被阻塞等待监视器锁定的线程处于此状态。
- WAITING  
正在等待另一个线程执行特定动作的线程处于此状态。
- TIMED\_WAITING  
正在等待另一个线程执行动作达到指定等待时间的线程处于此状态。
- TERMINATED  
已退出的线程处于此状态。

一个线程可以在给定时间点处于一个状态。 这些状态是不反映任何操作系统线程状态的虚拟机状态。



jdk 官方文档有 6 种状态，实际有 7 中，其中的 runnable 可以分为 ready 和 running 两个状态。

文件：

## ● 创建文件对象相关构造器和方法

### ➤ 相关方法

`new File(String pathname)` //根据路径构建一个File对象

`new File(File parent, String child)` //根据父目录文件+子路径构建

`new File(String parent, String child)` //根据父目录+子路径构建

### createNewFile 创建新文件



### ● 目录的操作和文件删除

- `mkdir` 创建一级目录、`mkdirs` 创建多级目录、`delete` 删除空目录或文件

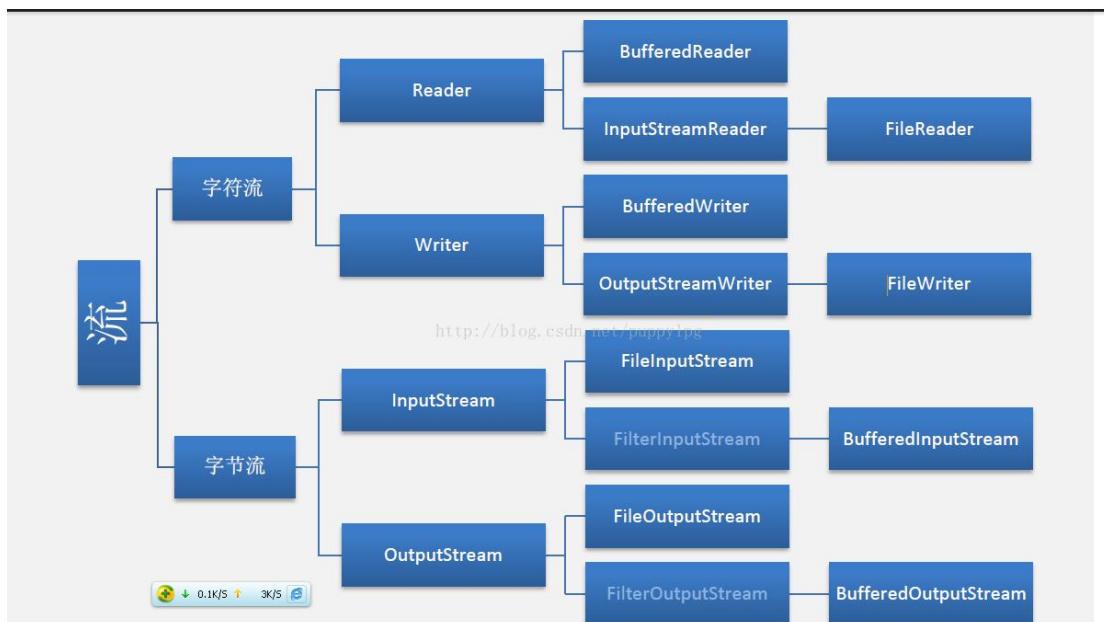
## ● 流的分类

- ✓ 按操作数据单位不同分为：字节流(8 bit) 二进制文件，字符流(按字符) 文本文件
- ✓ 按数据流的流向不同分为：输入流，输出流
- ✓ 按流的角色的不同分为：节点流，处理流/包装流



- 1) Java的IO流共涉及40多个类，实际上非常规则，都是从如上4个抽象基类派生的。
- 2) 由这四个类派生出来的子类名称都是以其父类名作为子类名后缀。

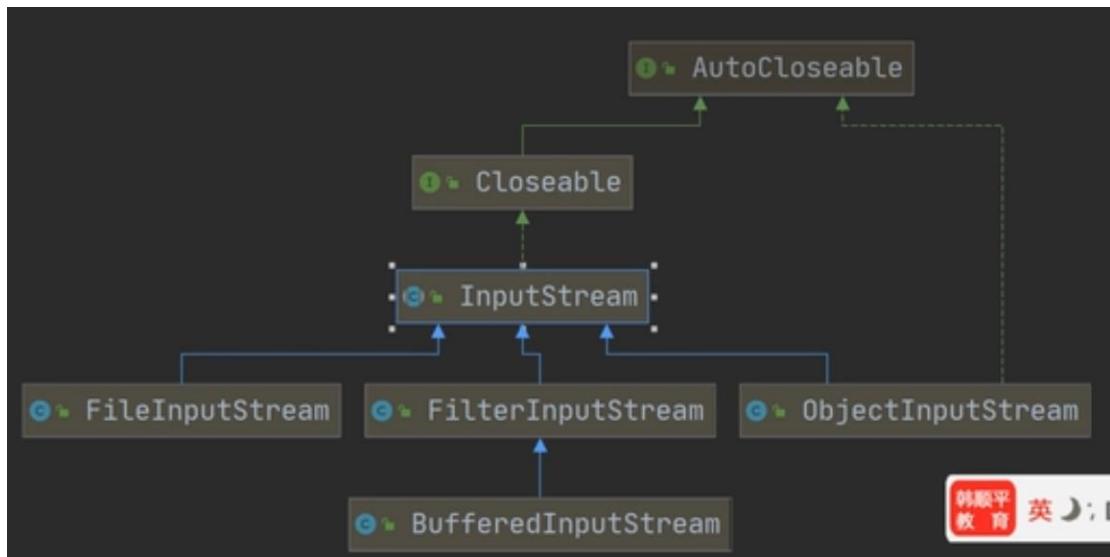
转自  
http://blog.csdn.net/pappy1pg



✓ **InputStream**抽象类是所有类字节输入流的超类

✓ **InputStream** 常用的子类

1. **FileInputStream**: 文件输入流
2. **BufferedInputStream**: 缓冲字节输入流
3. **ObjectInputStream**: 对象字节输入流



通过 `FileInputStream` 从文件中读取数据，如果该文件采用 UTF-8 编码，则输入流直接从磁盘中读取的就是数据编码后的二进制，读到内存中时，如果要打印到控制台，则将 `byte` 转换成 `char`，但是 java 中一个字符占两个字节，并且英文字符一个字节和两个字节的编码是一样的，强制转换不会有精度损失，但是汉字的 UTF-8 编码占 3 个字节，读取到一个字节后强转成 `char` 会出现乱码。

➤ 注意：  
**FileWriter使用后，必须要关闭(close)或刷新(flush)，否则写入不到指定的文件！**

```
//fileWriter.flush();
//关闭文件流，等价 flush() + 关闭
fileWriter.close();
```

| 分类    | 字节输入流                             | 字节输出流                              | 字符输入流                          | 字符输出流                           |
|-------|-----------------------------------|------------------------------------|--------------------------------|---------------------------------|
| 抽象基类  | <code>InputStream</code>          | <code>OutputStream</code>          | <code>Reader</code>            | <code>Writer</code>             |
| 访问文件  | <code>FileInputStream</code>      | <code>FileOutputStream</code>      | <code>FileReader</code>        | <code>FileWriter</code>         |
| 访问数组  | <code>ByteArrayInputStream</code> | <code>ByteArrayOutputStream</code> | <code>CharArrayReader</code>   | <code>CharArrayWriter</code>    |
| 访问管道  | <code>PipedInputStream</code>     | <code>PipedOutputStream</code>     | <code>PipedReader</code>       | <code>PipedWriter</code>        |
| 访问字符串 |                                   |                                    | <code>StringReader</code>      | <code>StringWriter</code>       |
| 缓冲流   | <code>BufferedInputStream</code>  | <code>BufferedOutputStream</code>  | <code>BufferedReader</code>    | <code>BufferedWriter</code>     |
| 转换流   |                                   |                                    | <code>InputStreamReader</code> | <code>OutputStreamWriter</code> |
| 对象流   | <code>ObjectInputStream</code>    | <code>ObjectOutputStream</code>    | <code>FilterReader</code>      | <code>FilterWriter</code>       |
| 抽象基类  | <code>FilterInputStream</code>    | <code>FilterOutputStream</code>    |                                |                                 |
| 打印流   |                                   | <code>PrintStream</code>           |                                | <code>PrintWriter</code>        |
| 推回输入流 | <code>PushbackInputStream</code>  |                                    | <code>PushbackReader</code>    |                                 |
| 特殊流   | <code>DataInputStream</code>      | <code>DataOutputStream</code>      |                                |                                 |

节点流  
处理流



```
//关闭流，这里注意，只需要关闭 BufferedReader，因为底层会自动的去关闭 节点流
//FileReader.
bufferedReader.close();
```

#### > 序列化和反序列化

1. 序列化就是在保存数据时，保存数据的值和数据类型
2. 反序列化就是在恢复数据时，恢复数据的值和数据类型
3. 需要让某个对象支持序列化机制，则必须让其类是可序列化的，为了让某个类是可序列化的，该类必须实现如下两个接口之一：
  - > Serializable // 这是一个标记接口, 没有方法
  - > Externalizable // 该接口有方法需要实现，因此我们一般实现上面的 Serializable 接口

1. 功能：提供了对基本类型或对象类型的序列化和反序列化的方法
2. ObjectOutputStream 提供 序列化功能
3. ObjectInputStream 提供 反序列化功能

```
//1. 读取(反序列化)的顺序需要和你保存数据(序列化)的顺序一致
//2. 否则会出现异常
```

- 1) 读写顺序要一致
- 2) 要求序列化或反序列化对象，需要 实现 Serializable
- 3) 序列化的类中建议添加SerialVersionUID,为了提高版本的兼容性
- 4) 序列化对象时，默认将里面所有属性都进行序列化，但除了static或transient修饰的成员
- 5) 序列化对象时，要求里面属性的类型也需要实现序列化接口
- 6) 序列化具备可继承性, 也就是如果某类已经实现了序列化，则它的所有子类也已经默认实现了序列化
 

加上SerialVersionUID，如果类中加了一个属性，则反序列化时不会认为你创建了一个新类，而会认为只是类的版本变化了

```

//System 类 的 public final static InputStream in = null;
// System.in 编译类型 InputStream
// System.in 运行类型 BufferedInputStream
// 表示的是标准输入 键盘
System.out.println(System.in.getClass());

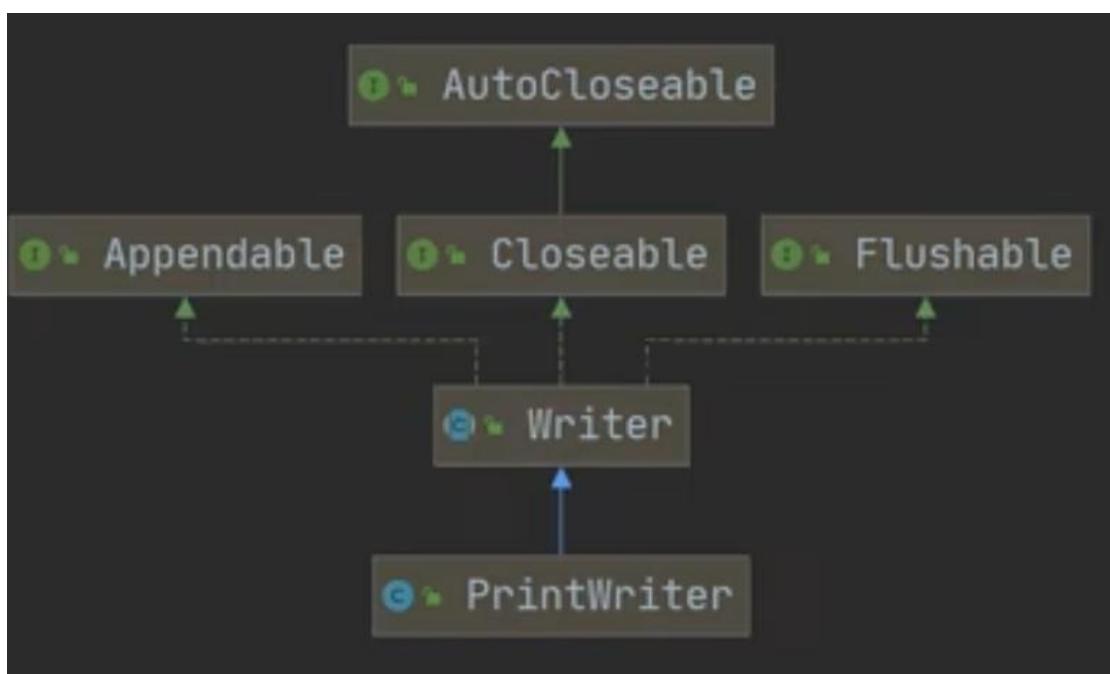
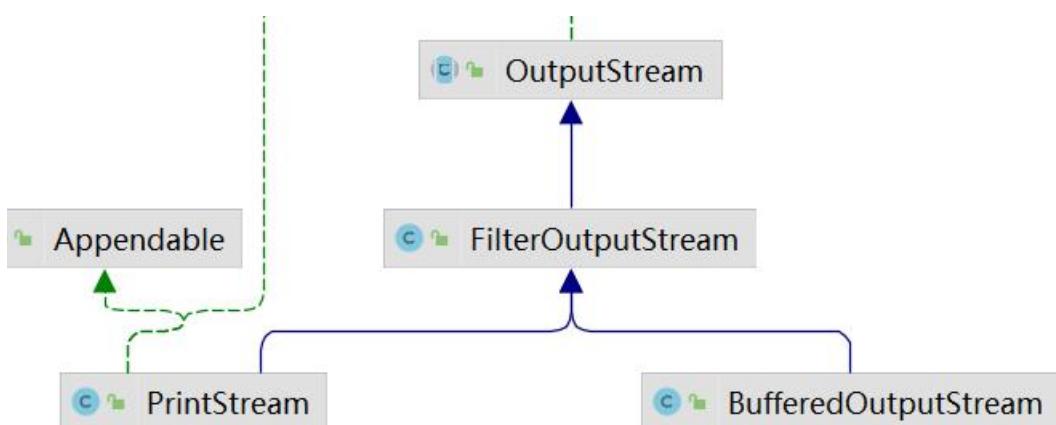
```

I

```

//老韩解读
//1. System.out public final static PrintStream out = null;
//2. 编译类型 PrintStream
//3. 运行类型 PrintStream
//4. 表示标准输出 显示器
System.out.println(System.out.getClass());

```



## ● 转换流-**InputStreamReader** 和 **OutputStreamWriter**

### ➤ 介绍

1. **InputStreamReader:Reader**的子类，可以将**InputStream(字节流)**包装成**Reader(字符流)**
2. **OutputStreamWriter:Writer**的子类，实现将**OutputStream(字节流)**包装成**Writer(字符流)**
3. 当处理纯文本数据时，如果使用字符流效率更高，并且可以有效解决中文问题，所以建议将字节流转换成字符流
4. 可以在使用时指定编码格式(比如 utf-8, gbk , gb2312, ISO8859-1 等)

读取文本文件时，文件编码和IDE编码方式不一样会出现乱码。而字符流不能处理，字节流可以处理，所以先通过字节流指定编码方式，然后将字节流转换成字符流。

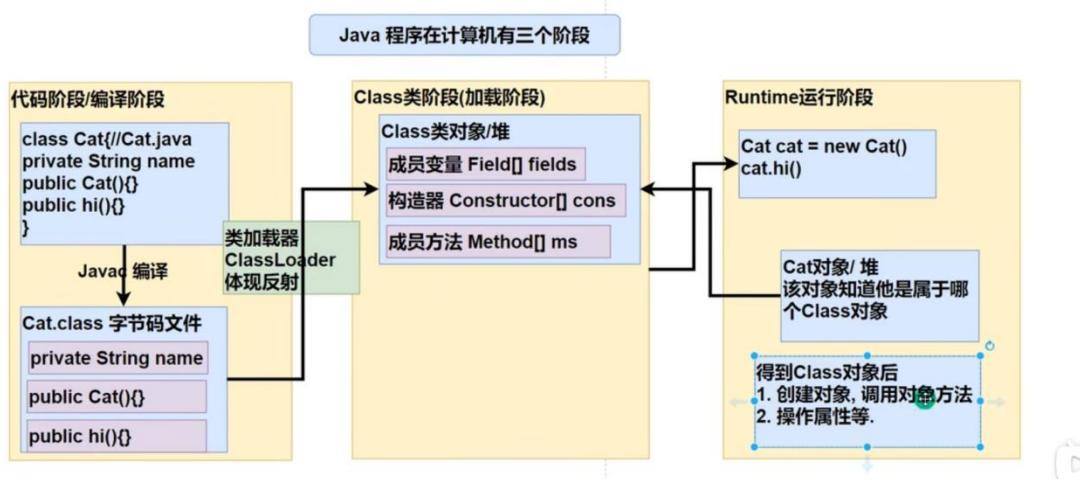
```
public |InputStreamReader| (@NotNull InputStream in, @NotNull Charset cs) {
```

字节输出流 flush()没有具体实现，字节流没有缓冲，而字符流有缓冲， flush()有具体实现

### 3) Properties的常见方法

- **load:** 加载配置文件的键值对到Properties对象
- **list:** 将数据显示到指定设备
- **getProperty(key):** 根据键获取值
- **setProperty(key,value):** 设置键值对到Properties对象
- **store:** 将Properties中的键值对存储到配置文件，在idea中，保存信息到配置文件，如果含有中文，会存储为unicode码

反射：



在类加载的第一个阶段(加载阶段)，会在堆中生成一个 Class 对象，这个 Class 类中的内部类中存储了这个类的属性，方法，构造器数组(这些都是当成对象看待的)，可以通过 Class 对象来动态地获得。

Class 对象不是 new 出来的，而是类加载器中的 loadClass 方法创建返回的。

## 反射机制

- Java反射机制可以完成

1. 在运行时判断任意一个对象所属的类
2. 在运行时构造任意一个类的对象
3. 在运行时得到任意一个类所具有的成员变量和方法
4. 在运行时调用任意一个对象的成员变量和方法
5. 生成动态代理

- Class类的常用方法:

| 方法名                                                    | 功能说明                               |
|--------------------------------------------------------|------------------------------------|
| static Class forName(String name)                      | 返回指定类名 name 的 Class 对象             |
| Object newInstance()                                   | 调用缺省构造函数，返回该Class对象的一个实例           |
| getName()                                              | 返回此Class对象所表示的实体（类、接口、数组类、基本类型等）名称 |
| Class [] getInterfaces()                               | 获取当前Class对象的接口                     |
| ClassLoader getClassLoader()                           | 返回该类的类加载器                          |
| Class getSuperclass()                                  | 返回表示此Class所表示的实体的超类的Class          |
| Constructor[] getConstructors()                        | 返回一个包含某些Constructor对象的数组           |
| Field[] getDeclaredFields()                            | 返回Field对象的一个数组                     |
| Method getMethod<br>(String name,Class ... paramTypes) | 返回一个Method对象，此对象的形参类型为paramType    |