

CH02基础

JDK(Java Development Kit)

JRE(Java Runtime Environment, JAVA运行环境)

针对使用者

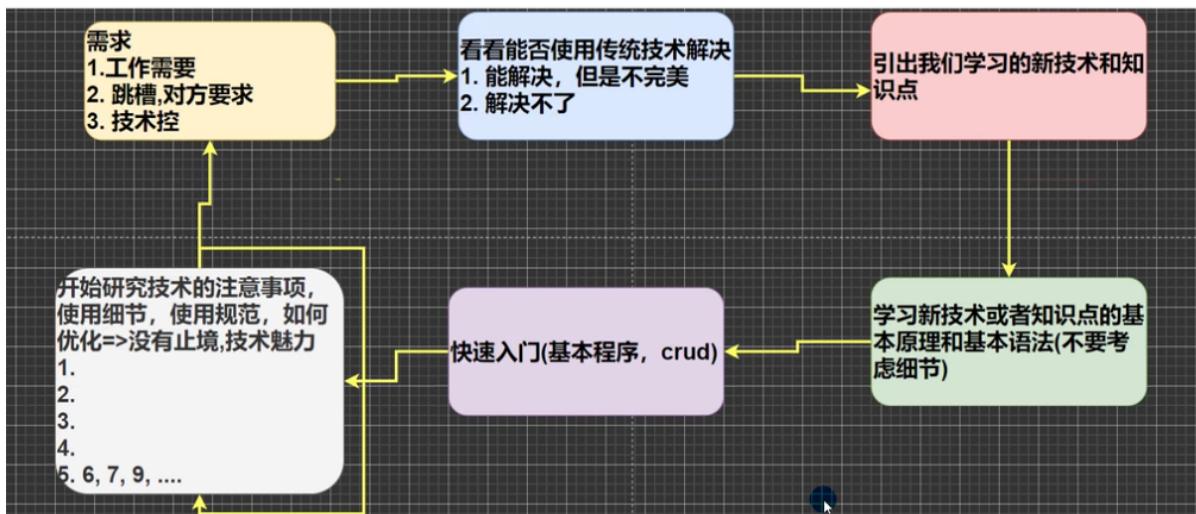
JVM

Java核心类库/Java SE标准类库

Java开发工具

针对开发者，包括

java, javac, javadoc, javap



javadoc

1. 类、方法的注释用文档注释的方式
2. 非javadoc注释的方式即代码内的单行或多行注释写给代码维护者看的

```
// .java源文件格式
// 文档注释
/**
 * @author yiga
 * @version 1.0
 */
// 生成java doc文件
javadoc -d w:code\\temp -author -version xx.java
```

行尾风格；次行风格



```
public ArrayList(int initialCapacity) {  
    if (initialCapacity > 0) {  
        this.elementData = new Object[initialCapacity];  
    } else if (initialCapacity == 0) {  
        this.elementData = EMPTY_ELEMENTDATA;  
    } else {  
        throw new IllegalArgumentException("Illegal Capacity: " +  
            initialCapacity);  
    }  
}
```

次行风格

DOS指令

```
// 返回上一级  
cd ..  
// 返回根目录  
cd \  
// 清屏  
cls  
// 查看所有子级目录  
tree  
// 查看当前目录  
dir  
// 创建文件夹  
md  
// 删除文件夹  
rd
```

java编写过程

1. 编写源代码.java
2. 使用DOS指令 javac 编译得到.class
3. 使用DOS指令 java 虚拟机运行.class

CH03 变量

浮点型

- 浮点型常量默认为double类型，8个字节，若使用float(4个字节)要先声明F

```
double num1 = 1.1 // 正确  
float num2 = 1.1 // 错误  
flaot num3 = 1.1F //正确  
double num4 = .1 // 正确 等价于1.1
```

- 允许使用科学计数法

```
double x = 1.1e-2;  
douoble x = 1.1e10;
```

- 小数是近似值，使用小数进行相等判断时要小心！

最好使用两个数的差值的绝对值在某个精度范围判断

```
double num1 = 2.7;
double num2 = 8.1 / 3;
output > num1=2.7, num2为接近2.7的一个小数2.69999999
Math.abs(num1 - num2) < 1e-6
```

字符串

- char的本质是整数，可进行数学运算，有Unicode对应的字符

```
char c1 = 'a';
// 输出的是97而不是a
System.out.println((int)c1);
char c2 = 97;
// 输出的是a，因此要输出使用(int)进行数据类型转换
System.out.println(c2);
// 可以进行数学运算，输出107
System.out.println('a' + 10);
```

字符编码表

ASCII编码表

1个字节=256位，实际只使用了128位

Unicode

字母和汉字都2个字节，浪费存储空间，共16位可编码65536个字符，兼容ASCII

utf-8

- 大小可变的编码，字母使用1个字节，汉字3个字节
- 变长的编码方式，使用1-6个字节表示一个符号

gbk

字母使用1个字节，汉字2个字节

数据类型转换

基本数据类型

数值型

整数型

byte[1], short[2], int[4], long[8]

byte: -128 ~ 127

short: -32768 ~ 32767

浮点型

float[4], double[8]

字符型

char[2]

布尔型

boolean[1]

精度低自动转为精度高的数据类型

- char -> int -> long -> float -> double
- byte -> short -> int -> long -> float -> double
- char和byte不自动转换，char和short不自动转换

```
byte b1 = 10; // 正确, 10是常量在byte范围内
int n2 = 1;
byte b2 = n2; // 错误, n2是int型变量, 占4个字节
char c1 = b1; // 错误, byte不与char转换
short s1 = 12;
char b3 = s1; // 错误, char不与short转换
-10.5 % 3 = ? // 输出1.5的近似值可能是1.500002
// 浮点型不能很好的表示准确的小数, 只是一个近似值
```

- byte, short, char可以计算, 计算时先转换为int类型

```
short s1 = 2;
byte b1 = 3;
short s2 = b1 + s1; // 错误, b2 + s1 => int
int n1 = b1 + s1; // 正确
s2 = s2 - 9; // 错误, 参与运算的short自动转为int
b1 = b1 + 10; // 错误, (b1 + 10) => 已经是int
int i = 3;
boolean b = (boolean) i; // 错误, Boolean不参与类型转换
```

- boolean类型不参与转换
- 自动提示原则：表达式结果类型自动提示为操作数最大的类型

强制类型转换

强制将精度高的转为精度低的

- 强制转换符号只与最近的操作数有效

```
int x = (int)10 * 3.5 + 2.3; // 错误, 只提升了10 * 3.5的数据类型
int x = (int) (10 * 3.5 + 2.3) // 正确
double d = 13.4;
long l = (long) d; // 正确
```

- char类型可以保护int常量值, 但无法保存变量值

```
int i1 = 100;
char c1 = i1; // 错误
char c1 = (char) i1; // 正确
System.out.println(c1); // 输出'd'
```

字符串类型转换

使用xx.parsexx() 的方法

唯独string不可以转换为char

```
String s = "3124";
int num1 = Integer.parseInt(s);
double num2 = Double.parseDouble(s);
float num3 = Float.parseFloat(s);
short num4 = Short.parseShort(s);
byte num5 = Byte.parseByte(s);
boolean num6 = Boolean.parseBoolean("true");
```

CH04 运算符

算数运算符

++ --

```
i = 1;
// temp = i, i = i + 1, i = temp;
i = i++; // 输出1
i2 = 1;
// i = i + 1, temp = i, i = temp
i2 = ++i2; // 输出2
```

关系运算符

短路与

逻辑运算符

&&

短路与，第一个条件为假则**不会**判断第二个条件，效率高；一般用短路与

&

逻辑与，第一个条件为假则第二个条件也会判断，效率低

如果第二个条件会改变变量则会大有不同

```
int a = 4;
int b = 9;
if (a < 1 && ++b < 50){
    ...
}
// 输出a = 4, b = 9
System.out.println("a = " + a "b = " + b);
if (a < 1 & ++b < 50){
    ...
}
// 输出a = 4, b = 10
System.out.println("a = " + a "b = " + b);
```

||

短路或，第一个条件为真则**不会**判断第二个条件，效率高；一般用短路或

|

逻辑或，第一个条件为真则第二个条件也会判断，效率低

```
int a = 4;
int b = 9;
if (a > 1 || ++b < 50){
    ...
}
// 输出a = 4, b = 9
System.out.println("a = " + a "b = " + b);
if (a > 1 | ++b < 50){
    ...
}
// 输出a = 4, b = 10
System.out.println("a = " + a "b = " + b);
```

!

逻辑取反

^

逻辑异或，一真一假判断为真

```
// 判断为真
boolean b = (10 > 1) ^ (3 < 2);
```

```
int x = 5;
int y = 5;
// x++ 先比较再自增
if(x++ == 6 & ++y == 6){
    x = 11;
}
// 输出x = 6, y = 6
int x = 5;
int y = 5;
```

```
if(x++ == 6 && ++y == 6){  
    x = 11;  
}  
// 输出x = 6, y = 5  
int x = 5, y = 5;  
if(x++ == 5 | ++y == 5){  
    x = 11;  
}  
// 输出 x = 11, y = 6  
int x = 5, y = 5;  
if(x++ == 5 | ++y == 5){  
    x = 11;  
}  
// 输出 x = 11, y = 5
```

三元运算符

条件? 表达式1:表达式2;

- 会自动转换类型，类型不匹配可使用强制类型转换
- 本质是if-else

```
int a = 10;  
int b = 99;  
// 输出result = 10  
int result = a > b ? a++: b--;
```

运算符优先级

单目运算符（针对一个位）运算顺序是从右向左的

```
. () {} ; ,  
// 单目运算符  
R->L ++ -- ~ !  
// 算术运算符  
* / %  
// 位移运算符  
<< >> >>> 位移  
// 比较运算符  
> < >= <= instanceof  
// 逻辑运算符  
== !=  
&  
^  
|  
&&  
||  
// 三元运算符  
?:  
// 赋值运算符  
R->L = *= /= %= += -= <=> >>= &= ^= |=
```

标识符规范

1. 包名：多单词组成时，所有字母均小写，aaa.bbb.ccc // torch.torch
2. 类名、接口名：多单词组成时，所有单词首字母大写 XxxYyyZzz[大驼峰]
3. 变量名、方法名：多单词组成时，第一个单词首字母小写，第二个单词开始首字母大写：xxxYyyZZZ[小驼峰]studentName
4. 常量名：所有字母大写，多单词连接时用下划线连接：XXX_YYY_ZZZ
5. 不能与关键字（均为小写）重名

二进制

原码、补码、反码

- 正数和0的原码、补码和反码都一样，没有变化，即**三码合一**
- 负数的反码=符号位不变，其他位取反
- 负数的补码 = 反码 + 1， 负数的反码 = 补码 -1
- Java没有无符号数，数都是有符号的，因而像short2个字节16位只能表示 $2^{16} / 2$ 的最大值
- 计算机运算都是以**补码**的方式来运算的
- 最终运算结果以**原码**方式呈现
- 最高位是符号位，正数是0，负数是1

位运算符

& | ^ ~ >> << >>>

按位与，按位或，按位异或，按位取反，

>>

算术右移，低位溢出，符号位不变，符号位补溢出的高位，本质是算术除以2

<<

算术左移，符号位不变，低位补0，本质是算术乘以2

>>>

无符号右移，低位溢出，高位补0

```
int a = 1 >> 2; // 00000000 00000000 00000000 00000000 01, 本质是 1 /2 / 2 =0
int c = 1 << 2; // 00000000 00000000 00000000 00000100, 本质是1*2*2=4
```

CH05 控制结构

分支结构

if 单分支

if else if 双分支

if else if else 多分支

嵌套分支

if里面再套if, 不要超过3层

switch

1. switch后面的表达式对应1个值
2. case 常量1表示当表达式的值等于常量1就执行语句1
3. break表示退出switch
4. case1后如果没有break会穿透到case2
5. 如果一个没有匹配上就执行default

```
switch(表达式){  
    case 常量1:  
        语句块1;  
        break;  
    case 常量2:  
        语句块2;  
        break;  
    ...  
    default:  
        语句块  
        break;  
}
```

while

while判断循环条件, 为真则继续执行, 为假直接退出

```
while(循环条件){  
}
```

do while

先执行一次循环再判断条件

```
do{  
}while(循环条件);
```

多重循环

循环套进另一个循环，一般使用两层，最多不要超过3层，否则代码可读性很差

continue

结束本次循环，继续执行下次循环

break

直接退出整个循环

return

跳出整个的方法

化繁为简 + 先死后活

```
输出
*
*
* *
*
*   *
*
*     *
*
*****
*****
```

化繁为简

```
1. 先输出
*****
*****
*****
*****
*****
2. 再输出
*
**
***
****
3. 再输出
*
* *
*   *
*****
*****
```

先死后活

有规律，找出了规律，常量改为变量

```
public class Stars{
    public static void main(String[] args){
        int i = 0;
        // 先死后活
        int totaltower = 30;
```

```

        while(i <= totaltower){
            int k = totaltower - i;
            int j = 2 * i + 1;
            while(k > 0){
                System.out.print(" ");
                --k;
            }
            while (j > 0){

                if(j == 2 * i + 1 || j == 1 || i == totaltower){
                    System.out.print("*");
                }
                else System.out.print(" ");
                --j;
            }
            System.out.println("");
            ++i;
        }
    }
}

```

```

*
*
* *
*
*   *
*
*   *
*
*   *
*
*   *
*
*   *
*****

```

CH06 数组

1. 数组是多个**相同类型数据**的组合，实现对数组的统一管理
2. 数组的元素可以包括引用类型，但不能混用得统一
3. 数组创建后没有初始时，有默认值

```

int 0, short 0, long 0, byte 0
float 0.0, double 0.0, char \u0000
boolean false, String null

```

4. 数组属于**引用类型**，数组型数据是对象

```

char[] c1 = {'a', 'b', 'c', 'd'};
char[] c2 = c1; // 指向的是地址而不是值！
c2[2] = 's';
// 输出结果c1: absd c2: absd

```

扩容方法

创建1个新的数组，然后赋值然后指针再赋值

String

1. 遍历String的方法

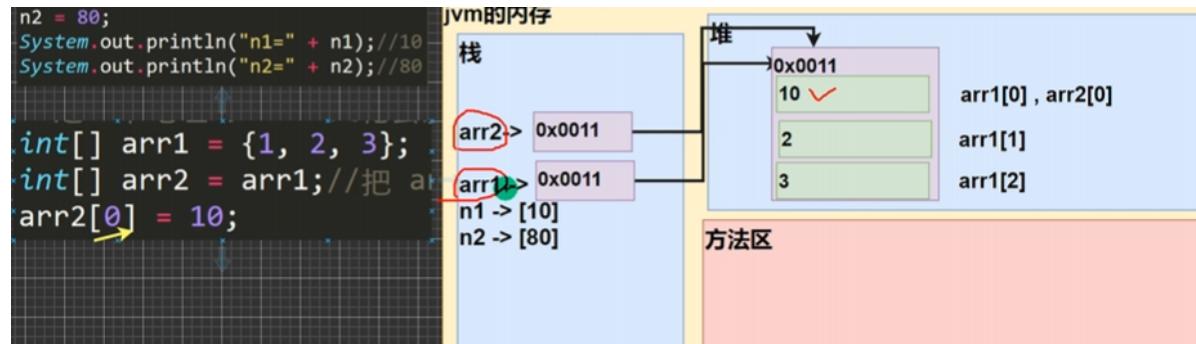
```
String s = " ssada";
// 把string转为char数组
for (Character c: s.toCharArray)
// 读取string的每个元素
for (int i = 0; i < s.length; ++i)
    char c = s.charAt(i);
```

2. 初始化String数组

```
String[] strs = {"a", "b", "c"};
String[] strs = new String {"a", "b", "c"}; // 错误，使用new创建对象要使用'[]'
String[] strs = new String[] {"a", "b", "c"};
String[] strs = new String[3] {"a", "b", "c"}; // 错误，使用new创建对象后面已经有
具体元素时，[]里不能写具体数
```

引用传递

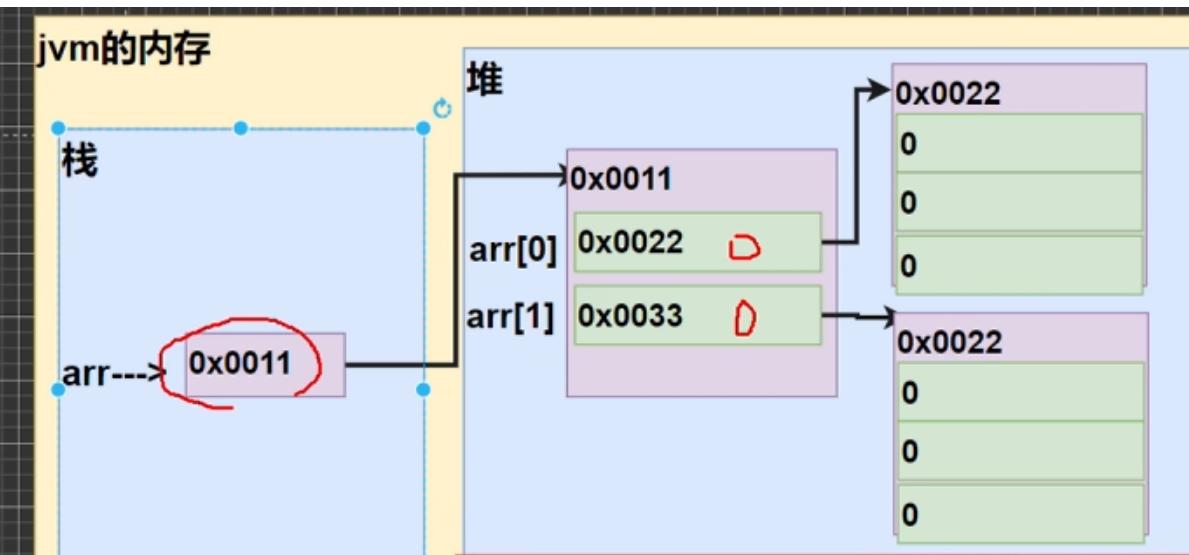
又称为地址传递，直接指向地址而不是数值



排序

二维数组

- 理解为指向指针的指针，数组的元素是地址



- 可以动态创建二维数组里的列，创建时保证列元素不要赋值

```
// 此处只是创建了行，但每个列的元素没有确定
int [][] arr = new int[3][];
for (int i = 0; i < arr.length; ++i){
    // 给第一列开了一个空间
    a[i] = new int[i+1];
    for (int j = 0; j < arr[i].length; ++j){
        a[i][j] = i + 1;
    }
}
// 最终arr的结果是1; 2, 2; 3, 3, 3;
```

- 静态初始化：使用'{}'创建，每一个{}为一维，二维数组里面的一维数组长度可以不同

```
int[][] arr= {{1,1,1},{8,8,9},{100}};
```

- 二维数组声明方式有很多

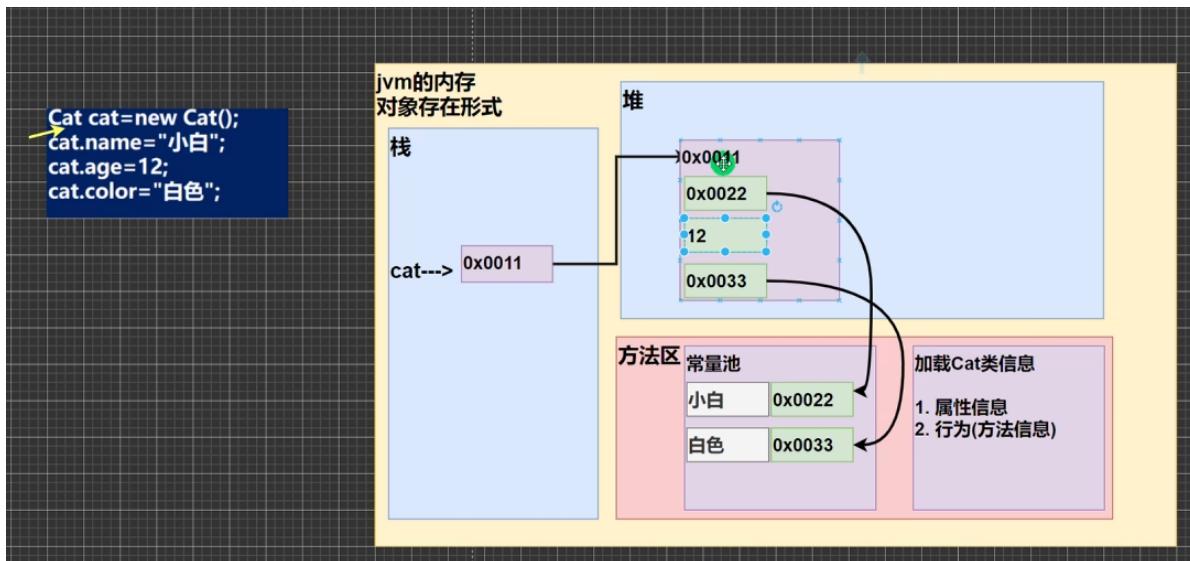
```
int[] x[] = new int[][]; // 也是二维数组
int[][] x
int x[][]
```

CH07 面向对象(初级)

对象的内存分配

- 栈：一般存放基本数据类型（局部变量）
- 堆：存放对象（数组，Person p）
- 方法区：常量池（常量，比如字符串），类加载信息

字符串常量会放在常量池



p1 = p2; 既可以说p1赋给了p2， 也可以说p2指向了p1

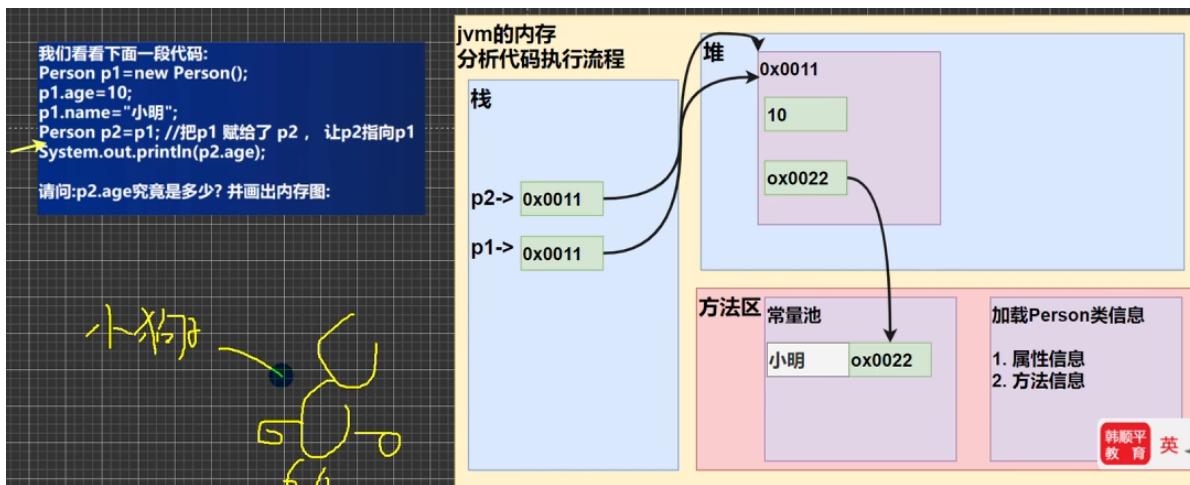
跟数组的赋值是一样的道理a2 = a1;赋予的是**对象的地址**

具体分配过程

1. 加载Person类信息 (属性和方法信息只会加载一次)

```
Person p = new Person();
p.name = "jack1";
p.age = 10;
```

2. 在堆中分配空间，进行默认初始化
3. 把地址赋给p, p就指向对象
4. 进行指定初始化，例如p.name = "jack1"



属性概念

属性 = 成员变量 = 字段field可以是基本数据类型，也可以是引用类型

1. 属性定义语法同变量，公式为：访问修饰符 属性类型 属性名

但多了访问修饰符：控制属性的**访问范围**，包含public private protected 默认

2. 属性有默认值

3. 对象名（对象引用）与对象不一样

```
// new Person是对象  
// p1是对象名  
Person p1;  
p1 = new Person();
```

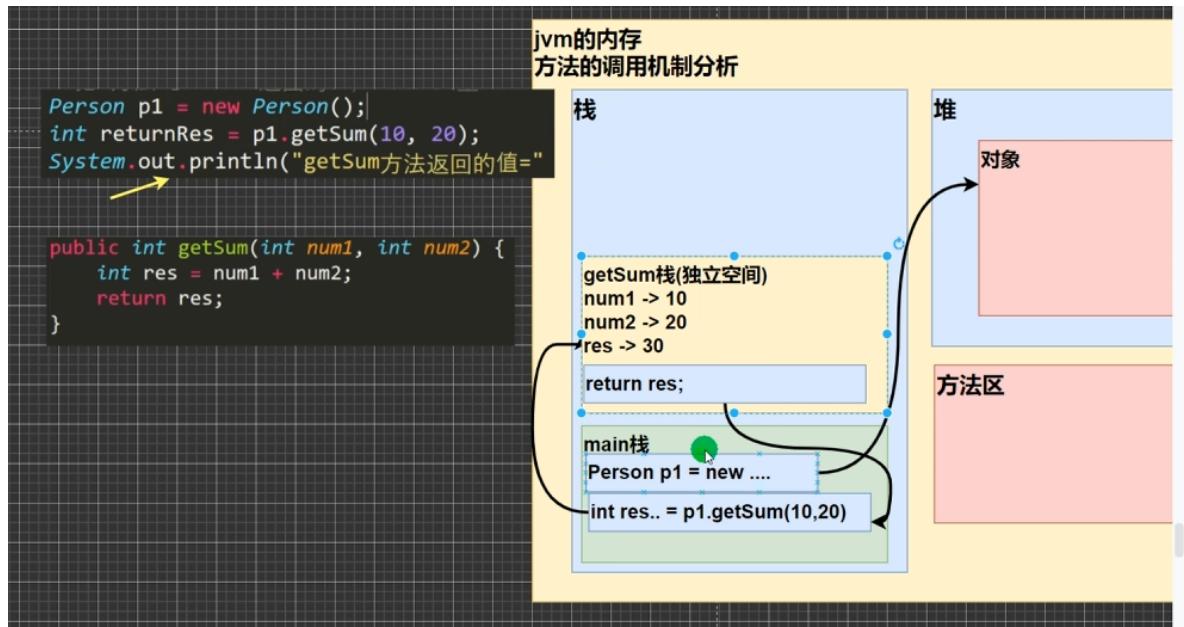
方法

特点

1. 提高代码的复用性
2. 将实现的细节**封装**起来，供其他用户调用

方法内存分配

1. 程序执行到方法时，会开辟一个独立的空间（栈空间）
2. 方法执行完毕或者执行完**return**后，因为getSum在栈创建的独立空间会被**销毁**，会返回调用方法的地方



原则

1. 一个方法最多有**1个**返回值，但返回类型和形参可以是任意类型，包含基本类型和**引用类型**（数组，对象）
2. 方法要求有返回数据类型，则方法最后的执行语句必须为**return值**
3. 命名遵循驼峰命名法
4. 方法定义时的参数是形参，调用时传入的是实参
5. 方法内不能**嵌套定义**其他方法
6. 调用：同一类的直接调用方法名；不同类的需要使用具体对象名.方法名

方法传参机制

1. 非引用类型的形参只传值不传地址，属于拷贝传递
2. 数组是引用类型，会指向堆里的空间，作为形参时是引用传递

The diagram shows the memory state during the execution of the code. It features two stacks: the main stack and the test200 stack. The main stack contains local variables `b` (pointing to a `B` object) and `arr` (pointing to a copy of the array [1, 2, 3]). The test200 stack contains local variables `p` (pointing to the same array [1, 2, 3]) and `b` (pointing to a `B` object). A red box highlights the shared array reference between the two stacks. The heap shows the array [1, 2, 3] and the `B` object. Handwritten annotations explain that both `p` and `arr` point to the same memory location in the heap.

```
B b = new B();
int[] arr = {1, 2, 3};
b.test100(arr); // 调用方法
System.out.println("main的 arr数组");
// 遍历数组
for(int i = 0; i < arr.length; i++) {
    System.out.print(arr[i] + "\t");
}
System.out.println();
}

class B {
    // B类中编写一个方法test100,
    // 可以接收一个数组，在方法中修改该数组，看看原来的数组是否变化
    public void test100(int[] arr) {
        arr[0] = 200; // 修改元素
        // 遍历数组
        System.out.println("test100的 arr数组");
        for(int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + "\t");
        }
    }
}
```

这是2个p，一个是main栈一个是test200栈的p，两个p互相独立虽然值的是同一个地址

The diagram illustrates object parameter passing. In the main stack, a `Person` object `p` is created with name "jack" and age 10. It is passed to the `test200` method in the `B` class. Inside the `test200` method, the `p` variable is set to `null`. The `main` stack still has its original `p` pointing to the `Person` object. The `test200` stack has its own `p` pointing to `null`. Handwritten annotations show the flow of `p` from the main stack to the method, and the modification of `p` in the method.

```

p.name = "jack";
p.age = 10;

b.test200(p);
// 测试题，如果 test200 执行的是 p = null，下面的结果是 10
System.out.println("main 的 p.age=" + p.age); // 10000
}

class Person {
    String name;
    int age;
}
class B {
    public void test200(Person p) {
        // p.age = 10000; // 修改对象属性
        // 思考
        p = null;
    }
}
```

方法重载

方法可同名，形参列表必须不一样，返回类型无所谓

可变参数

数据类型 + ... + 形参名字表示动态接收参数，作为数组使用

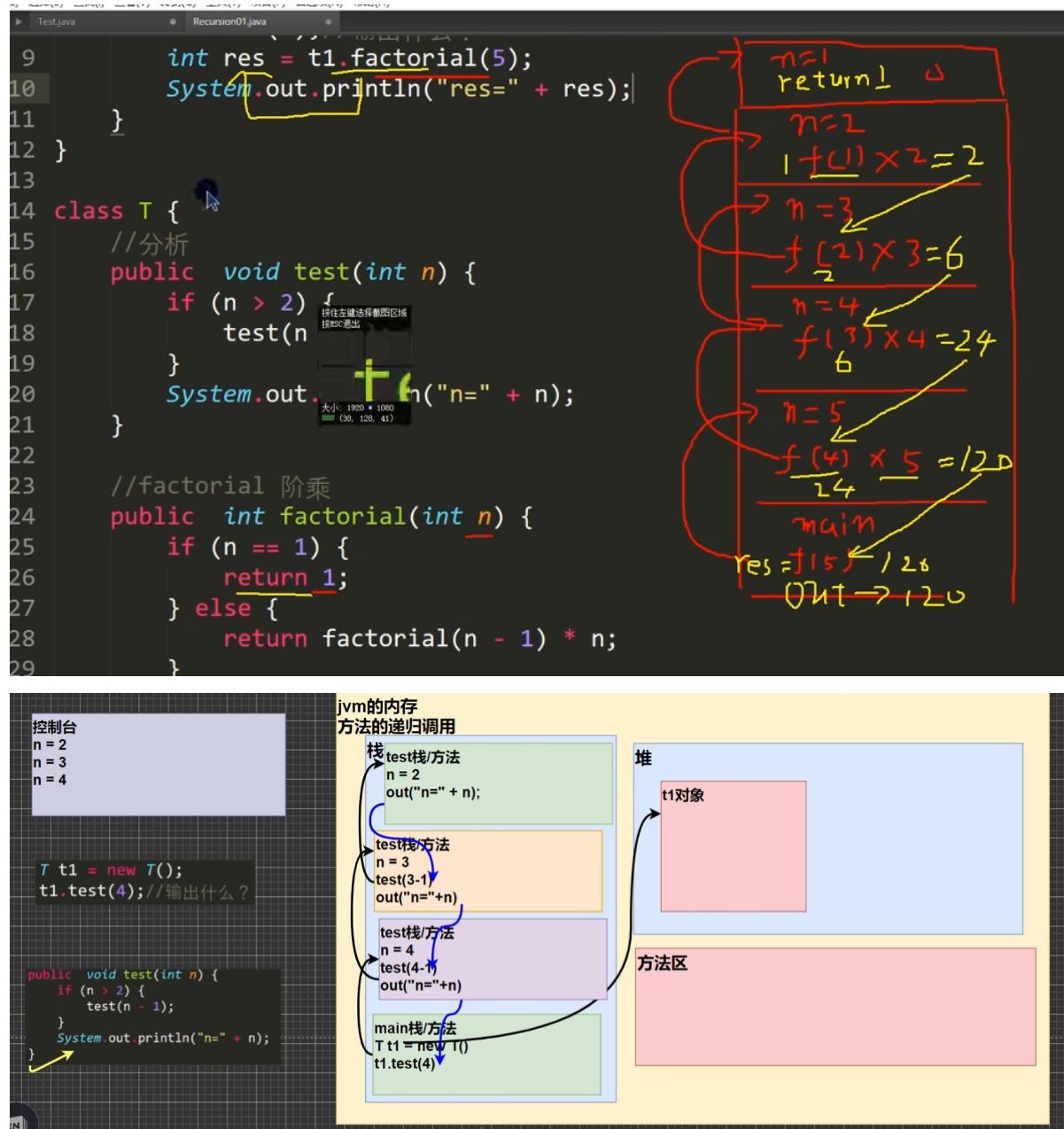
1. 可变参数本质就是数组
2. 可变参数实参是0或者任意多个
3. 可变参数的实参可以直接是数组
4. 一个形参列表只能由一个可变参数

```
public int sum (int ... nums)
public int sum (int ... nums, String s)
```

递归

简单的递归

阶乘



特性

- 每执行一次递归方法都创建一个受保护的**栈空间**，这个独立的栈空间内局部变量独立
- 递归必须要有终止条件，否则就是无限递归
- 一个方法执行完毕或遇到return，谁调用就返回给谁

练习

猴子吃桃

```
public int peach(int day){  
    if (day == 10){  
        return 1;  
    } else if (day >= 1 && day <=9){  
        return ((peach(day+1)+1)*2);  
    } else {  
        System.out.println("错误");  
        return -1;  
    }  
}
```

汉诺塔

如果是1层，则只要把A->C

如果是2层，则要把**前1层**移动到B，**第2层**移动到C，再前1层移动到C

如果是3层，则要把**前2层**移动到B，**第3层**移动到C，再前2层移动到C

.....因此要想解决n层则要解决n-1层，.....，一直递归下去

需要有把前n-1层**抽象成整体**的思想，不要太纠结于具体地递归过程

```
public void Hanoi(int num, char a, char b, char c){  
    if (num == 1){  
        // 直接移动到C  
        System.out.println(a + ' -> ' + c);  
    } else {  
        // 把前n - 1层移动到B  
        Hanoi(num - 1, a, c, b);  
        // 把第n层移动到C  
        System.out.println(a + ' -> ' + c);  
        // 把前n - 1层移动到C  
        Hanoi(num - 1, b, a, c);  
  
    }  
}
```

作用域(scope)

1. 局部变量一般指成员方法中的变量，作用域在定义它的代码块中
2. Java编程主要变量是属性（成员变量）和局部变量
3. 属性可以**不赋值**，直接使用因为有默认值；**局部变量必须赋值后才能使用**，因为没有默认值
4. 属性和局部变量可以重名，遵循**就近原则**
5. 同一个作用域（或成员方法）两个局部变量不能重名
6. 属性**生命周期长**，随对象生或死；局部变量生命周期短，随作用域生或死
7. 局部变量不可以被其他类使用，只能在本类对应方法中使用
8. 属性可以加修饰符，局部变量不行

构造器

```
[修饰符] 方法名(形参列表) {  
    方法体  
}
```

1. 没有返回值
2. 方法名=类名
3. 构造器调用由系统完成
4. 主要作用完成新对象的初始化
5. 可以实现构造器的重载
6. 若无定义构造器，会自动生成默认构造器
7. 一旦定义了构造器， 默认构造器就不存在

对象创建过程

```
class Person{  
    int age = 90;  
    String name;  
    Person (String n, int a){  
        name = n;  
        age = a;  
    }  
}  
Person p = new Person("小倩",20);
```

1. 在**方法区**加载Person.class，只会加载一次
2. 在堆中**分配对象空间**作为地址，即new Person，堆中有两个空间一个存放age一个存放name
3. 堆中**默认初始化**属性age = 0, name = null
4. **显式初始化**age = 90, name还是null
5. 执行构造方法完成**属性初始化**: name = "小倩" age = 20，在方法区的**常量池**中创建一个空间存放“小倩”；在堆中有一个地址指向方法区中的常量池的“小倩”
6. 完成对象初始化后，返回对象的地址，**p只是对象的引用或者对象名**，不是真正的对象，p指向堆的空间（**对象实际上在堆里面**）

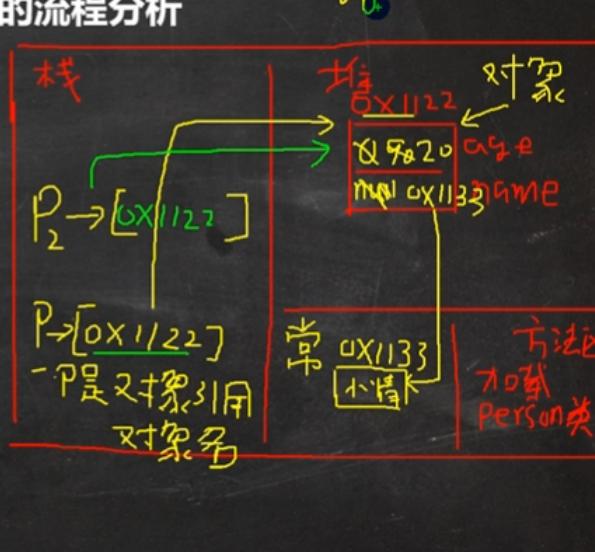
对象创建的流程分析

- 看一个案例

```
class Person{//类Person  
    int age=90; → 0  
    String name; → null  
    Person(String n,int a){//构造器  
        name=n;//给属性赋值  
        age=a;...  
    }  
    Person p=new Person("小情",20);
```

- 流程分析(面试题)

Person P₂ = P₁



this

代表当前对象，指的是对象本身在栈中存在的空间地址

- 用于区别类的属性和局部变量
- 不止是属性，还可以访问方法和构造器，访问方法 this.方法名
- 访问构造器时只能在构造器中使用，只能在一个构造器中访问另一个构造器，且必须放在第一条语句
- 必须在类内

```
class T{  
    public T(){  
        this("Jack", 100);  
        System.out.println("T()构造器");  
    }  
    public T(String name, int a){  
        System.out.println("第二个T()构造器");  
    }  
}
```

CH08 面向对象 (中级)

包

本质是创建不同文件夹来保存类文件

package关键词声明当前类所在的包，需放在类的最上面，一个类最多有一个package

import指令在package下面

命名规则

只包含数字、字母、下划线、小圆点，但不用数字开头，不能是关键字或保留字

命名规范

小写字母+小圆点

com.公司名.项目名.业务模块名

```
com.sina.crm.user // 用户名  
com.sina.crm.utils // 工具类
```

常用包

- java.lang.* 默认引入，不用再声明
- java.util.* 工具包
- java.net.* 网络包

引入包

建议使用哪个类就导入哪个类，避免使用*

```
import java.util.* // 引入所有类  
import java.util.Scanner // 只引入Scanner类
```

访问修饰符

1. public: 公开级别，对外公开
2. protected: 受保护级别，对子类和同一个包类公开
3. 默认级别：没有修饰符号，向同一个包的类公开
4. private: 私有级别，只有类本身可以访问

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	✓	✓	✓	X
4	默认	没有修饰符	✓	✓	X	X
5	私有	private	✓	X	X	X

- 不同包下，只能访问public不能访问protected 默认 private
- 同一个包下，不能访问private

注意事项

1. 可以修饰类中的属性，成员方法和类
2. 只有默认和public才能修饰类

封装(encapsulation)

步骤

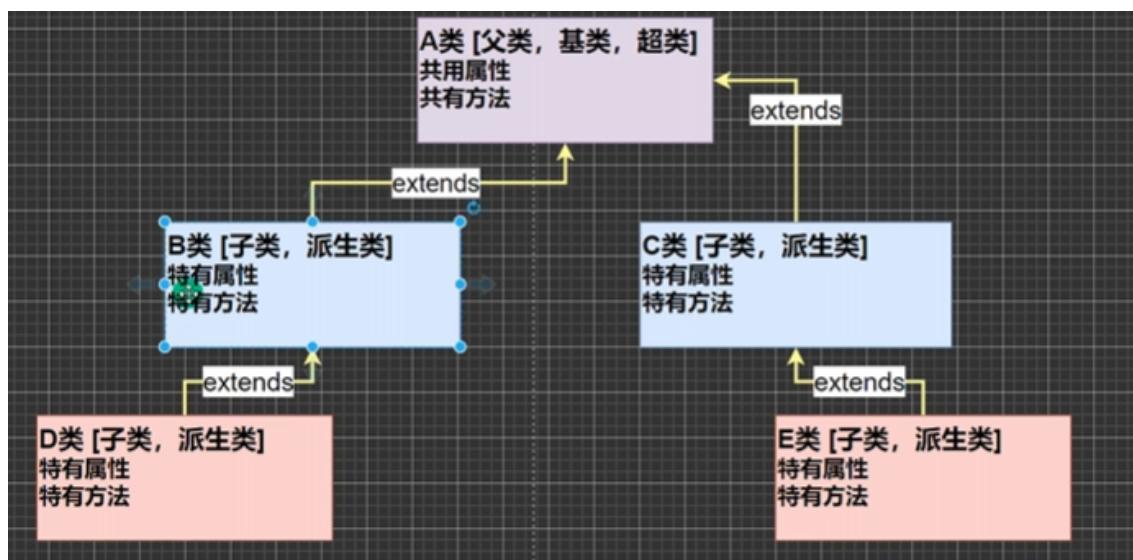
1. 属性私有化private (不能直接修改属性)
2. 提供一个public void set方法，用于对属性判断并赋值
3. 提供一个public 数据类型 get方法，用于获取属性的值

构造器可以和封装联动，把set写到构造器里面去，所有参数初始化都是用set方法而不是直接this.变量赋值

```
public Person(String name, int age, double salary) {  
    setName(name);  
    setAge(age);  
    setSalary(salary);  
}
```

继承(extends)

- 父类又叫超类、基类，子类又叫派生类



细节

1. 子类继承所有属性和方法，但**private**属性和方法不能在子类直接访问，要通过public方法访问；非**private**属性和方法可以在子类直接访问
2. 子类必须调用父类的构造器完成父类的初始化
3. 创建子类对象时，不管使用子类哪个构造器默认情况总会隐藏一个super调用**父类无参构造器**
4. 如果父类没有提供无参构造器，则在子类必须用显示使用**super**指定父类的某个构造器完成初始化工作
5. super在使用时，要放在子类构造器的第一行，且只能在构造器中使用，并且父类构造器先调用再子类
6. super和this必须在构造器中使用，但两者不能同时存在
7. Java所有类都是Object类的子类，**Object类**是始祖类尤尼尔
8. 父类构造器的调用不限于直接父类，将一直往上追溯到Object类（顶级父类）

9. 子类最多只能直接继承一个父类
10. 不能滥用继承，要满足is - a关系 (person extends music , person is a music? no!)
11. 在构造器中若没有super或者this那么会**隐藏的在第一行有一个super!**

super

代表父类的引用，用于访问父类的属性、方法、构造器

1. 能访问父类的属性、方法和构造器，但不能访问父类的private属性和方法 super.属性名 super.方法

细节

1. super专门用于父类初始化
2. 不用本类而**专门调用父类方法可以用super**，super不加时调用方法遵循**就近原则**（会一直查找到Object类）
3. **this.方法**和什么都不加方法是等价的
4. 属性中this和super使用与方法同理
5. super访问不限于直接父类，可以使用父父类，但遵循就近原则

● super和this的比较

No.	区别点	this	super
1	访问属性	访问本类中的属性，如果本类没有此属性则从父类中继续查找	从父类开始查找属性
2	调用方法	访问本类中的方法，如果本类没有此方法则从父类继续查找。	直接访问父类中的方法
3	调用构造器	调用本类构造器，必须放在构造器的首行	调用父类构造器，必须放在子类构造器的首行
4	特殊	表示当前对象	子类中访问父类对象

重写/覆盖(override)

@override

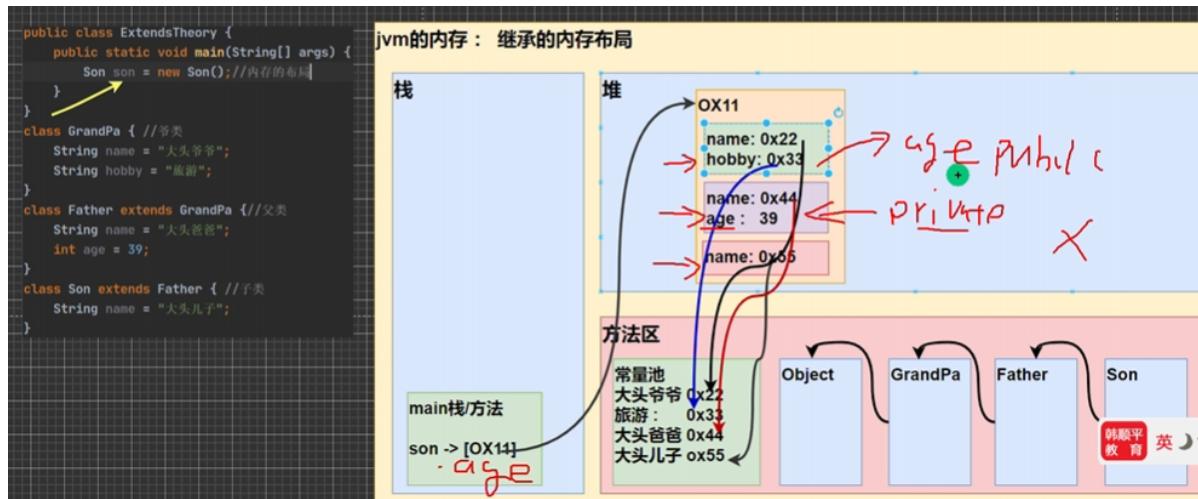
1. 子类中重写后的方法**名称、形参列表**与父类一样
2. 要求返回类型必须与父类返回类型一样，或者是父类返回类型的**子类**
例如父类返回Object，子类返回String
3. 子类方法**不能缩小**父类的访问权限 public > protected > 默认 > private，但可以扩大
4. **属性**没有重写之说

```
public void m1(){}
// 子类中的m1
// 错误，不行
void m1(){}
```

与重载(overload)区别

1. 发生范围：一个在本类；一个父子类
2. 方法名：都必须一样
3. 形参列表：一个必须不同；一个必须相同
4. 返回类型：一个无要求；一个必须相同类或其子类
5. 修饰符：一个无要求；一个只能扩大或等于访问权限

内存分配



1. Object -> GrandFa -> Father -> Son
2. 访问属性时，若Son没有则会看Father有没有，遵循就近原则

本质

继承的本质是建立一个查找关系

多态

建立在封装和继承基础之上，OOP的第三大特征，方法或对象具有多种状态

1. 一个对象的编译类型和运行类型可以不一致

```
Animal animal = new Dog(); // 编译类型是Animal, 运行类型是Dog  
animal = new Cat(); // 编译类型不变仍然是animal, 运行类型是Cat
```

2. 编译类型在定义对象时就确定了不能改变
3. 运行类型可以改变
4. 定义时“=”左边是编译类型，右边是运行类型，编译类型可以指向运行类型是子类的对象

表现

1. 方法多态：重载；重写
2. 对象多态：编译类型与运行类型可以不一致，**编译类型定义时就确定不可变**
运行类型可以变，可以通过getClass()查看运行类型

向上转型

父类的引用指向子类的对象

```
Animal animal = new Dog(); // 编译类型是Animal, 运行类型是Dog
public Person{
    public void eat() {}
    public void run() {}
}
public Student extends Person{
    public void eat() {}
    public void say() {}
}
Person p = new Student(); // 向上转型, 父类引用指向子类对象
p.eat(); // 根据动态绑定, 执行的是Student类的eat方法
p.say(); // 错误, 不能访问子类特有成员
p.run(); // 正确
Student s = (Student) p; // 子类的引用指向了(指向子类)的父类引用
s.eat(); // 执行的是Student类的eat方法
s.run();
s.say();
```

- 可以调用父类的所有成员属性和方法
- 但**不能调用子类特有成员**，因为在**编译阶段**能调用哪些成员是由**编译类型**决定的，可以通过向下转型调用子类特有成员
- 运行阶段，最终运行效果要看子类具体实现，即当父类和子类都有相同的方法时，查找方法从**本类**开始找

向下转型

```
Animal animal = new Cat(); // 本来animal指向的就是Cat对象, 即运行类型是Cat
Cat cat = (Cat) animal; // 编译类型是Cat, 运行类型是Cat, animal的编译类型是Animal
Dog dog = (Dog) animal; // 错误, 因为animal指向的不是dog类型
```

1. 只能强转父类的引用，不能强转父类的对象
2. 要求父类的引用必须指向的是**当前目标类型的对象**
3. 向下转型后，可以调用子类类型的所有成员包括特有方法，本质是运行类型变了

属性重写

- 属性没有重写之说，只看编译类型
- instanceof: 比较操作符，用于判断对象的运行类型是否为XX类型或XX类型的子类型

```
cat instanceof Animal
```

```
public class PolyDetail02 {  
    public static void main(String[] args) {  
        //属性没有重写之说！属性的值看编译类型  
        Base base = new Sub(); //向上转型  
        System.out.println(base.count); // ? | 看编译类型 10  
    }  
}  
  
class Base { //父类  
    int count = 10; //属性  
}  
  
class Sub extends Base { //子类  
    int count = 20; //属性  
}
```

输出count = 10;

动态绑定

1. 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. 调用对象属性时，没有动态绑定，哪里声明哪里调用

```
class A { //父类  
    public int i = 10;  
    public int sum() {  
        return getI() + 10;  
    }  
    public int sum1() {  
        return i + 10;  
    }  
    public int getI() {  
        return i;  
    }  
}  
  
class B extends A { //子类  
    public int i = 20;  
    public int sum() {  
        return getI() + 20;  
    }  
    public int getI() {  
        return i;  
    }  
    public int sum1() {  
        return getI() + 10;  
    }  
}  
  
//main方法中  
A a = new B(); //向上转型  
System.out.println(a.sum()); //?40  
System.out.println(a.sum1()); //30  
2min
```

- java的动态绑定机制
- 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
- 当调用对象属性时，没有动态绑定机制，哪里声明，那里使用

1. 假设B里面没有sum()函数：a.sum()会先根据动态绑定看运行类型B里面有沒有sum函数，若沒有再查找父类的a.sum(),然后getI() + 10,调用对象属性时,沒有动态绑定因此此时的i取A中的i = 10,结果为20

2. 假设B里面有sum1函数，则直接调用getI() + 10 = 30

多态数组

创建编程类型为父类，运行类型为可变子类的数组

```
Person[] persons = new Person[5];
persons[0] = new Student("JackyLove", 100);
persons[1] = new Teacher("Tian", 3000);
persons[2] = new Student("JackyLove2", 100);
persons[3] = new Teacher("Tian2", 3000);
persons[4] = new Student("JackyLove3", 100);
for (int i = 0; i < persons.length; i++) {
    System.out.println(persons[i].say());
    if (persons[i] instanceof Student){
        ((Student) persons[i]).study();
    } else if (persons[i] instanceof Teacher){
        ((Teacher) persons[i]).teach();
    } else {
        System.out.println("类型有误差");
    }
}
```

多态参数

同多态数组，编译类型是大父类，运行类型是子类

```
Worker tom = new Worker("Tom", 2000);
Managger jack = new Managger("Jack", 3000, 4000);
test.testwork(tom);
// Employee e是编译类型，根据运行类型不同走分支
public void testwork(Employee e) {
    if (e instanceof Worker){
        ((Worker) e).work();
    } else if (e instanceof Managger) {
        ((Managger) e).manage();
    }
}
```

Object类

方法

equals

- 在Object类下只能用于判断引用类型
- 在子类如String Integer时会重写用于判断属性是否相等

```
Integer i1 = new Integer(5);
Integer i2 = new Integer(5);
System.out.println(i1 == i2); // false, 比较的是地址
System.out.println(i1.equals(i2)); // true, 比较的是内容
```

```

Person p1 = new Person();
p1.name = "hspedu";

Person p2 = new Person();
p2.name = "hspedu";

System.out.println(p1==p2);
System.out.println(p1.name.equals(p2.name));
• System.out.println(p1.equals(p2));

String s1 = new String("asdf");

String s2 = new String("asdf");
System.out.println(s1.equals(s2));
System.out.println(s1==s2);
3min

```

```

class Person{//类
public String name;
}

```

F (地址不同) T (内容相同) F (地址不同) T (内容相同) F (地址不同)

重写equals用于比较对象的内容是不是相同

```

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj instanceof Person) {
        // 关键是创建一个中间量用于向下转型
        Person p = (Person) obj;
        return this.name.equals(p.name) && this.age == p.age;
    }
    return false;
}

```

==

比较运算符

1. 既可以判断基本类型，又可以判断引用类型
2. 基本类型：判断值是否相等
3. 引用类型：地址是否相等

```

//代码如下 EqualsExercise03.java
int it = 65;
float fl = 65.0f;
System.out.println( "65和65.0f是否相等? " + (it == fl));
char ch1 = 'A' ; char ch2 = 12;
System.out.println( "65和 'A' 是否相等? " + (it == ch1));
System.out.println( "12和ch2是否相等? " + (12 == ch2));

String str1 = new String("hello");
String str2 = new String("hello");
System.out.println("str1和str2是否相等? " + (str1 == str2));

System.out.println( "str1是否equals str2? " +(str1.equals(str2)));
System.out.println( "hello" == new java.sql.Date());

```

T(内容相等) T (内容相等) T (内容相等) F(引用类型不等) T(String.equals.内容相等) 报错 (类型不等)

hashCode

用于提高哈希结构的容器的效率

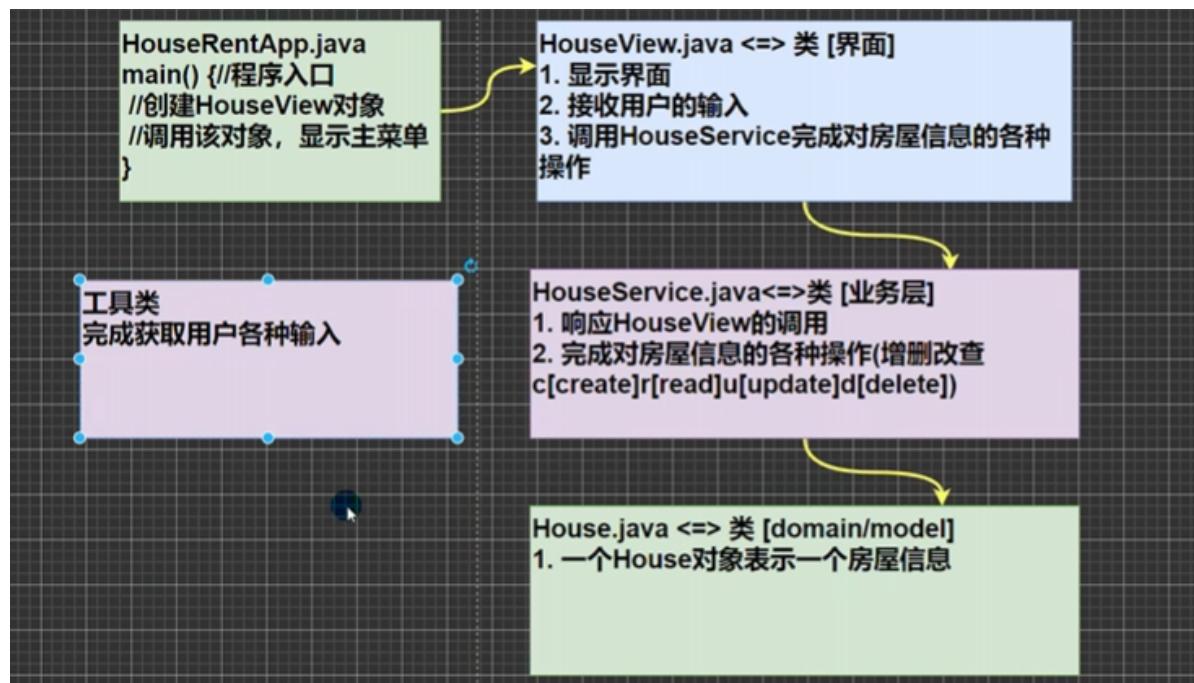
1. 两个引用，如果指向同一个对象，则返回的哈希值相同；反之，则不一样
2. 哈希值根据地址，但Java中在虚拟机上得不到真正的地址，因此哈希值不等价于地址
3. 哈希值根据地址号

toString(Obj)

返回(String): 全类名 (包名+类名) +@+哈希值的16进制

finalize

小project



CH09 面向对象 (高级)

类变量、类方法

访问修饰符 + static + 数据类型/返回类型 + 变量名/方法名；

类变量 (静态变量) 细节

1. 与实例变量区别：是否有static？是否被所有对象共享？
2. 非静态变量称为实例变量/普通变量/非静态变量
3. 访问静态变量推荐使用**类名.类变量**

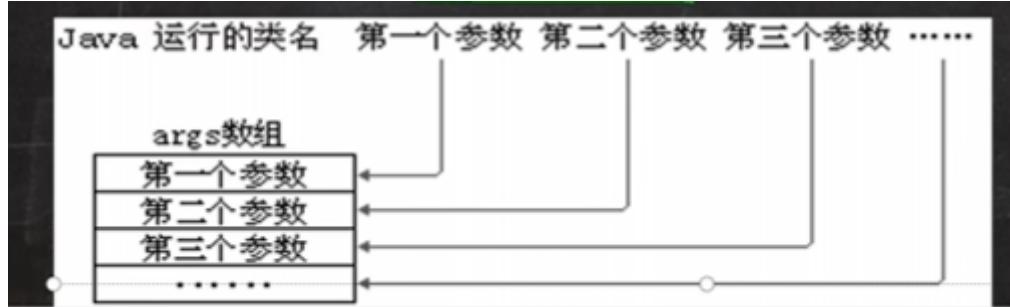
类方法（静态方法）细节

1. 静态只能访问静态成员（方法和属性），静态只能访问静态（太被动了）
2. 普通成员方法，既可以访问非静态成员，也可以访问静态成员
3. 可以使用类名.方法名或者对象.方法名，普通方法不能用类名.方法名调用
4. 不涉及任何和对象相关的成员，如果不希望创建实例也可以调用某个方法（当作工具来使用），比如Utility工具类，Math数学类
5. 与普通方法一样，都是随着类的加载而加载，将结构信息存储在方法区中
6. 不允许使用和对象有关的关键字，比如this和super

main方法

```
public void main(String[] args)
```

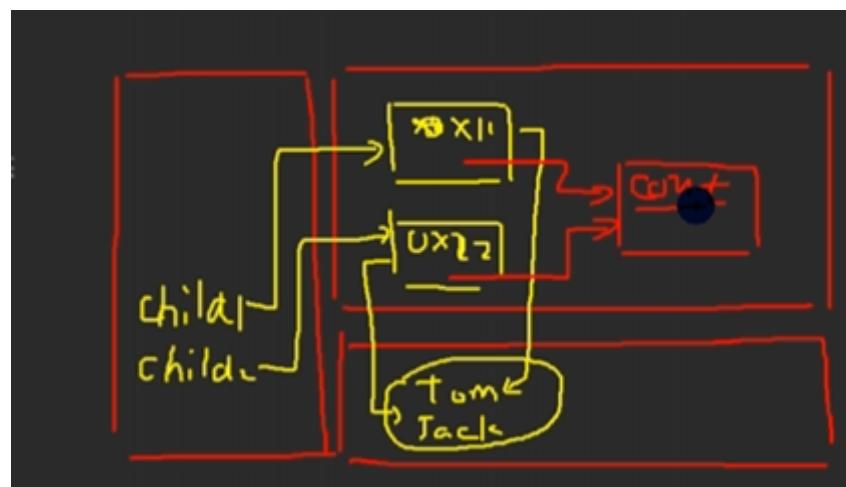
1. 由Java虚拟机(jvm)调用，虚拟机不在同一个类，因此要public
2. jvm调用main方法不需创建对象，因此使用static
3. 字符串数组args在程序运行过程中传入 (java xx arg1 arg2 arg3)



4. 在main以内不能访问本类的非静态成员！
5. 若要访问本类的非静态成员，需要先实例化本类的对象再调用

内存分配

类变量放在堆中，被所有类的对象共享，再类加载的时候就生成了，jdk7以前是在方法区的静态域中加载



代码块

```
[修饰符] {  
    代码  
};
```

1. 修饰符可选，要写只能写static
2. 代码块分为2类：static修饰的**静态代码块**；没有static修饰的普通代码块/非静态代码块
3. ';'可省可不省
4. 另外形式的构造器，**对构造器的补充**，做初始化操作
5. 多个构造器有重复语句，可以抽取到代码块中提高代码**复用性**

细节

1. **静态代码块**对类进行初始化，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一次**
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）

```
public class CodeBlockDetail01 {  
    public static void main(String[] args) {  
        AA aa = new AA(); // 创建子类对象时，此时已加载BB 和 AA静态代码块  
        BB bb = new BB(); // 不再加载静态代码块  
        System.out.println(cat.n1); // 使用类的静态成员也会加载静态代码块  
    }  
}  
class Cat{  
    public static int n1 = 999;  
    static {  
        System.out.println("静态代码块cat1被执行");  
    }  
}  
class BB {  
    static {  
        System.out.println("静态代码块B1被执行");  
    }  
}  
class AA extends BB{  
    static {  
        System.out.println("静态代码块A1被执行");  
    }  
}
```

3. 普通的代码块，在**创建对象实例**时，会被**隐式的调用**；**对象被创建一次，就会被调用一次**；跟类是否被加载无关；**构造器**被调用，它就被调用

4. 创建对象时，在一个类的调用顺序是：静态>普通>构造器（父类静态>子类静态>父类普通>父类构造>子类普通>子类构造）

1. 调用静态代码块和静态属性初始化时，两者优先级一样，先到（先定义）的先调用

```
A a = new A();
// 先调用getN1 再是静态代码块
class A {
    // getN1先定义
    private static int n1 = getN1();
    static {
        System.out.println("A 静态代码块01");
    }
    public static int getN1() {
        System.out.println("getN1被调用");
        return 100;
    }
}
B b = new B();
// 先调用静态代码块，再是getN1
class B {
    static {
        System.out.println("A 静态代码块01");
    }
    private static int n1 = getN1();
    public static int getN1() {
        System.out.println("getN1被调用");
        return 100;
    }
}
```

2. 调用普通代码块和普通属性初始化，两者优先级一样，同样是先到先得

3. 调用构造器

```
A a = new A();
// 调用顺序是：
// 1.静态代码块
// 2.静态方法getN1()
// 3.普通方法getN2()
// 4.普通代码块
// 5.构造器
class B {
{
    System.out.println("B 代码块");
}
public B() {
    System.out.println("B 构造器");
}
}
class A extends B{
public A(){
    System.out.println("A 构造器");
}

private int n2 = getN2();
```

```

private static int n1 = getN1();
{
    System.out.println("A 普通代码块01");
}
static {
    System.out.println("A 静态代码块01");
}
public static int getN1() {
    System.out.println("A getN1");
    return 100;
}
public int getN2() {
    System.out.println("A getN2");
    return n2;
}
}

```

5. 构造器最前面隐含了super和调用本类的普通代码块；静态相关的代码块，属性初始化，在类加载时，就执行完毕，因此优先于构造器和普通代码块

6. 有继承关系时总顺序

1. 父类的静态代码块和静态属性（优先级一致，按定义顺序执行）
2. 子类的静态代码块和静态属性（优先级一致，按定义顺序执行）
3. 父类的普通代码块和普通属性初始化（优先级一致）
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
6. 子类的构造方法

```

new Dog();
class Animal{
    private static int n1 = getval01(); // (1)
    private int n3 = getval03(); // (5)
    static {
        System.out.println("父类静态代码块"); // (2)
    }
    {
        System.out.println("父类普通代码块"); // (6)
    }
    public static int getval01(){
        System.out.println("父类静态方法初始化静态属性");
        return 33;
    }
    public int getval03(){
        System.out.println("父类普通方法初始化普通属性");
        return 33;
    }

    public Animal() {
        System.out.println("父类构造器"); // (7)
    }
}

class Dog extends Animal{

```

```

    static {
        System.out.println("子类静态代码块"); // (3)
    }
    {
        System.out.println("子类普通代码块"); // (8)
    }
    private static int n2 = getval02(); // (4)
    private int n4 = getval04(); // (9)

    public static int getval02(){
        System.out.println("子类静态方法初始化静态属性");
        return 22;
    }
    public static int getval04(){
        System.out.println("子类普通方法初始化普通属性");
        return 33;
    }
    public Dog() {
        // 隐藏了super和普通代码块
        System.out.println("子类构造器"); // (10)
    }
}

```

7. 静态代码块（本质是静态方法）只能直接调用静态成员，普通代码块可以调用任意成员

练习

平教育

代码块

题2：下面的代码输出什么？

CodeBlockExercise02.java

```

class Sample
{
    Sample(String s)
    {
        System.out.println(s);
    }
    Sample()
    {

        System.out.println( "Sample默认构造函数被调用");
    }
}

class Test{
    Sample sam1=new Sample("sam1成员初始化");//静态成员sam初始化
    static{
        System.out.println("static块执行");//如果sam==null)System.out.println("sam is null");
    }
    Test()
    {
        System.out.println("Test默认构造函数被调用");
    }
}
//主方法
public static void main(String str[])
{
    Test a=new Test();//无参构造器
}

```

顺序：

1. Test类加载，静态属性sam，静态代码块
2. Test类创建对象，普通属性，构造器

单例设计模式

在软件执行过程中，对某个类只能存在一个对象实例，并提供一个取得对象实例的方法

饿汉式(类加载时已被创建好)

1. 构造器私有化，防止直接new
2. 类的内部创建static对象
3. 提供一个public的static方法，返回类的对象

```
System.out.println(GirlFriend.getInstance());  
  
class Girlfriend {  
    private String name;  
    private static Girlfriend gf = new Girlfriend("wuwu");  
    private Girlfriend(String name) {  
        this.name = name;  
    }  
    public static Girlfriend getInstance(){  
        return gf;  
    }  
}
```

懒汉式 (调用方法时才创建对象)

1. 构造器私有化
2. 定义一个static对象
3. 定义一个public的static方法，返回类对象
4. 只有当用户使用上述方法时才创建类对象，后面再次调用时，会返回上次创建的对象，保证单例，避免内存调用

```
class Girlfriend {  
    private String name;  
    // 区别在此处没有直接创建对象，而只是定义对象  
    private static Girlfriend gf;  
    private Girlfriend(String name) {  
        this.name = name;  
    }  
    public static Girlfriend getInstance(){  
        if (gf == null){  
            gf = new Girlfriend("wuwu");  
        }  
        System.out.println(gf.toString());  
        return gf;  
    }  
}
```

区别

1. 懒汉式存在线程安全，饿汉式存在资源浪费

final

可以修饰类、方法、属性和局部变量

1. 不希望类被继承
2. 不希望父类某个方法被子类覆盖/重写
3. 不希望类的某个属性被修改
4. 不希望某个局部变量被修改

细节

1. final修饰的属性又称为常量，一般用大写XX_XX_XX_XX来命名
2. final修饰的属性在类加载时，必须赋初值后不再可被修改，复制位置有：
 1. 定义时
 2. 构造器中
 3. 代码块中
3. 如果final修饰的属性是静态的，则初始化位置有：
 1. 定义时
 2. 静态代码块，不能在构造器因为必须在类加载时
4. 一般一个类如果已经是final，方法不需要再用final
5. final不能修饰构造器
6. final和static往往搭配使用，效率更高，不会导致类加载
7. 包装类（String, Integer, Double等）都是final类，不可被继承

抽象类

- 当父类的一些方法不确定时，用abstract关键词来修饰该方法
- 当一个类中存在abstract方法，需要把整个类声明成abstract类
- 一般来说，抽象类会被继承，由其子类来实现抽象方法
- 抽象方法没有方法体，访问修饰符 + abstract + 返回类型 + 方法名（参数列表；）

细节

1. 抽象类不能被实例化
2. 抽象类不一定有抽象方法
3. abstract只能修饰类和方法，不能修饰属性和其他
4. 抽象类可以有任意成员，其本质还是类
5. 如果一个类继承了抽象类，则必须实现（有方法体）抽象类的所有抽象方法，除非自己也声明为abstract类
6. 抽象方法不能用private、final和static来修饰，因为与override相违背

模板使用模式

设计一个父类用定义抽象方法，子类再重写

```

public class AbstractDetail02 {
    public static void main(String[] args) {
        AA aa = new AA();
        aa.calculateTime();
        BB bb = new BB();
        bb.calculateTime();
    }
}

public abstract class Template { // 抽象类模板
    public abstract void job(); // 抽象方法
    public void calculateTime() {
        long start = System.currentTimeMillis();
        job(); // 动态绑定机制
        long end = System.currentTimeMillis();
        System.out.println("计算时间 " + (end - start));
    }
}

class AA extends Template{
    @Override
    public void job(){
        long num = 0;
        for (int i = 0; i < 1000000; i++) {
            num += 1;
        }
    }
}

class BB extends Template{
    @Override
    public void job(){
        long num = 0;
        for (int i = 0; i < 1000000; i++) {
            num *= 1;
        }
    }
}

```

接口

给出没有实现的方法封装到一起，到某个类要使用的时候，根据具体情况把这些方法写出来

- jdk7.0以前接口所有方法都没有方法体即都为**抽象方法**
- jdk8.0以后接口可以有静态方法，**默认方法**，即接口有方法的**具体实现**；默认方法需要使用**default**关键字修饰，静态方法使用**static**修饰即可
- 接口中，抽象方法可以**省略abstract**关键词

```
interface 接口名{
    // 属性
    // 方法（1.抽象方法；2.默认实现方法；3.静态方法）
}

class implements 接口名{
    // 自己属性
    // 自己方法
    // 必须实现的接口的抽象方法
}
```

细节

1. 接口不能被**实例化**, 本质是**抽象类**
2. 接口所有的方法都是隐含的带了**public**, abstract可以省略不写
3. 普通类接口要将接口所有方法都**实现**; 抽象类实现接口则不需要实现
4. 一个类可以同时**实现多个接口**, 与继承只能单继承不同
5. 接口中的**属性只能是final**, int a 实际上是 **public static final int a**, 其属性的访问方式是**接口名.属性名**
6. 接口**不能继承其他的类**, 但可以**继承多个接口**
7. 接口修饰符**只有public和默认**, 跟类的修饰符一样

```
interface A{
    int a=23; //等价 public static final int a = 23;
}
class B implements A{//正确
}
main中:
B b=new B();//ok
System.out.println(b.a); //23
System.out.println(A.a); //23
System.out.println(B.a); //23
```

都正确, 均可以访问public static final属性

与继承的区别

接口是对单继承机制的一种补充

1. 继承价值: 解决代码**复用性和可维护性**
2. 接口价值: 设计好各种**规范方法**, 让其它类去实现这些方法
3. 接口比继承更灵活, 不需要满足is-a的关系, 像**like-a**
4. 在一定程度上实现**代码解耦 (接口规范性+动态绑定)**



```
public class ExtendsVsInterface {  
    public static void main(String[] args) {  
        LittleMonkey wukong = new LittleMonkey("wukong");  
        wukong.climbing();  
        wukong.swimming();  
        wukong.flying();  
    }  
}  
  
class LittleMonkey extends Monkey implements Fishable, Birdable{  
    public LittleMonkey(String name) {  
        super(name);  
    }  
  
    @Override  
    public void swimming() {  
        System.out.println(getName() + "通过学习学会了游泳");  
    }  
  
    @Override  
    public void flying() {  
        System.out.println(getName() + "通过学习学会了飞行");  
    }  
}  
  
class Monkey{  
    private String name;  
    public void climbing(){  
        System.out.println(getName() + "爬树");  
    }  
  
    public Monkey(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}  
  
interface Fishable{  
    public void swimming();  
}  
interface Birdable{
```

```
    public void flying();
}
```

多态特性

1. 其多态特性与继承一致，同款向上转型，**运行类型**可以与编译类型不一致
2. 多态数组也是与继承同款
3. 接口**多态传递**现象

```
main(){
    IG ig = new IH(); // 向上转型
    IG ig2 = new AA(); // IH继承了IG, AA实现了IH; 因此AA也可以是实现了IG
}
interface IG{}
interface IH extends IG{}
public class AA implements IH{}
```

接口与父类的变量相同时，要调用时要明确是谁的（接口的用类名.变量名，父类的用super.变量名）

```
public class InterfaceExercise02 {
    public static void main(String[] args) {
        new B().printX();
    }
}

interface i1{
    int x = 0;
}
class A {
    int x = 1;
}
class B extends A implements i1{
    public void printX(){
        System.out.println("i1的x是" + i1.x + " A的x是" + super.x);
    }
}
```

内部类

一个类的内部嵌套了另一个类结构，**被嵌套的类**称为内部类(inner class)，最大特点是可以**直接访问私有属性**，体现类与类之间的包含关系

类的五大成员：**属性、方法、构造器、代码块、内部类**

分类

- 定义在外部类局部位置（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

1. 成员内部类 (没有static修饰)

2. 静态内部类 (用static修饰)

局部内部类

1. 可以直接访问**外部类所有成员**, 包含private
2. **不能添加访问修饰符**, 但是可以用final修饰, 因为**本质是局部变量类**, 局部变量也可以用final, 但用了final后**不能再被其他内部类继承**
3. 作用域只在**定义它的方法体或代码块中**
4. 外部类在方法中可以创建**局部内部类对象**, 然后调用方法
5. **外部其他类不能访问**局部内部类
6. 如果外部类和局部内部类的**成员重名**, 遵循**就近原则**, 如果要访问外部类成员, 则可以使用**外部类名.this.成员**去访问

```
public class LocalInnerClass {  
    public static void main(String[] args) {  
        Outer01 outer01 = new Outer01();  
        outer01.m1();  
    }  
}  
class Outer01 { // 外部类  
    private int n1 = 20;  
  
    private void m2() {  
        System.out.println("私有方法m2");  
    }  
  
    public void m1() {  
        class Inner01 { // 局部内部类  
            private int n1 = 800;  
            public void f1() {  
                System.out.println("局部内部类的n1 = " + n1);  
                // Outer01.this本质就是外部类的对象, 即哪个对象调用了m1, Outer01.this就是哪个对象  
                // main方法中是outer01调用了方法f1因此Outer01.this指的就是outer01  
                System.out.println("外部类的n1 = " + Outer01.this.n1);  
                m2();  
            }  
        }  
        Inner01 inner01 = new Inner01();  
        inner01.f1();  
    }  
}
```

匿名内部类

本质仍然是类; 还是在内部里; 该类没有名字; 同时是一个对象

1. 基本语法: **new 类或接口(参数列表) {类体};**

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

```
        Outer04 outer04 = new Outer04();
        outer04.method();
    }
}

class Outer04{
    private int n1 = 10;
    public void method(){
        Tiger tiger1 = new Tiger();
        tiger1.cry();
        // 使用匿名内部类简化:不创建tiger
        // tiger2的编译类型是II
        // 运行类型是匿名内部类
        /*
        * 底层会分配类名Outer04$01
        * class Outer04$01 implements II{
            @Override
            public void cry() {
                System.out.println("老虎叫唤");
            }
        }*/
        // jdk底层在创建匿名内部类Outer04$01，立即创建Outer04$01的实例，并把地址返回给
        tiger2
        // 匿名内部类使用一次后就不能再使用
        II tiger2 = new II() {
            @Override
            public void cry() {
                System.out.println("老虎叫唤");
            }
        };
        System.out.println("tiger2的运行类型是" + tiger2.getClass());
        tiger2.cry();
        // 编译类型是A01，运行类型是Outer04$2，相当于
        /* class Outer04$2 extends A01{
            @Override
            public void test() {
                System.out.println("匿名内部类"+this.getClass()+"重写了test方
法");
            }
        };*/
        // 仍然遵循类的加载，会启动父类的构造器
        A01 jklove = new A01("jklove"){
            @Override
            public void test() {
                System.out.println("匿名内部类"+this.getClass()+"重写了test方
法");
            }
        };
        System.out.println("A01对象的运行类型 = " + jklove.getClass());
        jklove.test();
    }
}

interface II{
    public void cry();
}

// 要创建类必须实现II接口很生硬，称之为“硬编码”，不推荐
```

```

class Tiger implements II{
    @Override
    public void cry() {
        System.out.println("老虎叫唤");
    }
}
class A01{
    public A01(String s) {
        System.out.println("构造器初始化" + s);
    }

    public void test(){
    }
}

```

2. 匿名内部类本身是类，同时又返回类的实例化对象

```

public class AnnoymousInnerClassDetail {
    public static void main(String[] args) {
        Outer05 outer05 = new Outer05();
        outer05.f1();
    }
}

class Outer05{
    private int n1 = 99;
    public void f1(){
        Person p = new Person(){
            @Override
            public void hi() {
                System.out.println("匿名内部类重写了hi方法");
            }
        };
        // 返回类的实例化对象
        p.hi();
        // 也可以直接调用，匿名内部类本身也返回对象
        new Person(){
            @Override
            public void hi() {
                System.out.println("匿名内部类又重写了hi方法2");
            }
            public void hello(){
                System.out.println("匿名内部类独有的hello方法");
            }
        }.hi();
        new Person(){
            @Override
            public void hi() {
                System.out.println("匿名内部类又重写了hi方法2");
            }
            public void hello(){
                System.out.println("匿名内部类独有的hello方法");
            }
        }.hello();
    }
}

class Person{

```

```
public void hi(){
    System.out.println("Person hi()");
}
```

3. 可以直接访问外部类的所有成员包含私有的
4. **不能添加访问修饰符**, 本质是**局部变量**, 同局部内部类
5. 作用域在定义它的**方法体和代码块中**
6. 外部其他类不可以直接访问匿名内部类 (局部变量)
7. 外部类成员和匿名内部类变量重名, 就近原则, 外部类成员访问用 **外部类名.this.成员**
8. 作为实参传递更加简洁高效

成员内部类

定义在外部类的成员位置上

1. 可以直接访问外部类的所有成员, **包括私有**
2. 可以添加任意访问修饰符, 因为它的**地位就是一个成员**
3. 作用域为整个类体
4. 外部类访问成员内部类, 需要先实例化对象
5. **外部其他类**访问成员内部类:
 - 先创建外部类对象, 再 **外部类实例化对象名.new 成员内部类名**
 - 使用一个方法返回成员内部类的实例

```
public class MemberInnerClass01 {
    public static void main(String[] args) {
        Outer08 outer08 = new Outer08();
        outer08.t1();
        // 方法1
        // 把new Inner08()当作是outer08的成员
        Outer08.Inner08 inner08 = outer08.new Inner08();
        // 方法2
        Outer08.Inner08 inner081 = outer08.getInner08Instance();
        // 方法3
        // 本质与法1相同
        Outer08.Inner08 inner082 = new Outer08().new Inner08();
    }
}

class Outer08{
    class Inner08{
        public void say(){
            System.out.println("成员内部类");
        }
    }
    public void t1(){
        Inner08 inner08 = new Inner08();
        inner08.say();
    }
    public Inner08 getInner08Instance(){
        return new Inner08();
    }
}
```

```
    }  
}
```

6. 外部类成员与内部类成员同名：就近原则；外部类名.this.成员

```
// 成员内部类使用案例，需要用到外部类成员  
public class Homework07 {  
    public static void main(String[] args) {  
        Car car = new Car(-3);  
        car.getAir().flow();  
    }  
}  
  
class Car{  
    private int temperature;  
    class Air{  
        public void flow(){  
            if (temperature > 40){  
                System.out.println("冷风");  
            } else if (temperature < 0) {  
                System.out.println("暖风");  
            }  
        }  
    };  
    public Car(int temperature) {  
        this.temperature = temperature;  
    }  
    public Air getAir(){  
        return new Air();  
    }  
}
```

静态内部类

定义在外部类的**成员位置**，有static修饰

1. 可以访问外部类的所有**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，本质就是一个成员
3. 作用域为整个类体
4. 外部类访问静态内部类成员，同样是创建对象再访问
5. 外部其它类访问静态内部类，使用 **外部类名.静态内部类**
6. 外部类和静态内部类成员重名时，就近；外部类名.this.成员

枚举和注解

枚举(enumeration)

枚举是一组**常量的集合**，可以理解为特殊的类（包含一组有限的特定的对象）

自定义实现

1. 不需要提供setXXX方法，因为枚举对象通常为只读
2. 对枚举对象/属性使用**final+static**共同修饰，实现底层优化
3. 枚举对象名通常全部大写

```
class Season {  
    private final String name;  
    // 定义了4个对象  
    public static final Season SPRING = new Season("春天");  
    public static final Season SUMMER = new Season("夏天");  
    public static final Season AUTUMN = new Season("秋天");  
    public static final Season WINTER = new Season("冬天");  
  
    private Season(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```

关键字实现

1. 使用**enum**关键字代替class
2. 常量名1(实参列表),常量名2实参列表),常量名3(实参列表),...来初始化（逗号隔开）；注意**常量要放在第一行**

```
enum Season2 {  
    // 常量对象写在最前面  
    // 等价于public static final Season SPRING = new Season("春天");  
    SPRING("春天"), SUMMER("夏天"), AUTUMN("秋天"), WINTER("冬天"), SUMMER2;  
    private String name;  
  
    private Season2(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
    Season2() {  
    }  
}
```

细节

1. 使用enum关键字开发枚举类时，**默认会继承Enum类**，因此不能再继承其他类，而且是**final类**
2. 使用无参构造器创建枚举对象时，**实参列表和小括号都可以省略**，如案例中的SUMMER2
3. enum实现的枚举类仍是类，**可以实现接口**

成员方法

1. `toString()`: 在`System.out.println()`中使用对象名会隐式调用
2. `name()`: 返回枚举对象名
3. `ordinal()`: 返回对象的位置序号，默认从0开始
4. `values()`: 返回枚举类的所有常量，组成一个**多态数组**
5. `valueOf()`: 返回指定常量名的常量
6. `compareTo()`: 比较两个枚举常量的编号是否相等

```
public class Enumeration01 {  
    public static void main(String[] args) {  
        Season2 s = Season2.SUMMER;  
        System.out.println("s.name = " + s.name());  
        System.out.println("s.ordinal = " + s.ordinal());  
        // s.values()返回枚举对象并集合成一个数组  
        System.out.println("s.values =" + s.values());  
        for (Season2 season2 : s.values()) {  
            System.out.println("遍历取出枚举对象" + season2);  
        }  
        // 返回指定名称的枚举对象  
        System.out.println("Seanson2.valueOf = " + Season2.valueOf("AUTUMN"));  
        // 比较其编号(ordinal)是否相等  
        System.out.println("s.compareTo(Season2.SUMMER)" +  
s.compareTo(Season2.SUMMER));  
    }  
}
```

注解(Annotation)

@override

重写父类方法

1. **编译层面验证**，不写也没事，但不能乱写（无中生有会报错）
2. 只能修饰方法，不能修饰其他类、包、属性

`override`是一个**注解类**(@interface，不是接口)

```
// @Target是修饰注解的注解，称为元注解  
@Target(ElementType.METHOD)  
@Retention(RetentionPolicy.SOURCE)  
public @interface override {  
}
```

@Deprecated

某个程序元素（类、方法）已经过时，但仍然可以使用

- 可以修饰方法，类，字段，包，参数等
- 做到新旧版本的兼容和过渡

```
@Documented  
@Retention(RetentionPolicy.RUNTIME)  
@Target(value={CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER,  
TYPE})  
public @interface Deprecated {  
}
```

@SuppressWarnings

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})  
@Retention(RetentionPolicy.SOURCE)  
public @interface SuppressWarnings {  
    String[] value();  
}
```

抑制编译器警告

语法：@SuppressWarnings("警告名")

- 作用范围与放置位置有关
- unchecked：没有检查的警告；rawtypes：没有指定泛型的警告；unused：没有使用某个变量的警告

```
public class SuppressWarnings_ {  
    @SuppressWarnings({"unchecked", "unused", "all"})  
    public static void main(String[] args) {  
        List list = new ArrayList();  
        list.add("ss");  
        list.add("w");  
        System.out.println(list.get(1));  
    }  
}
```

元注解

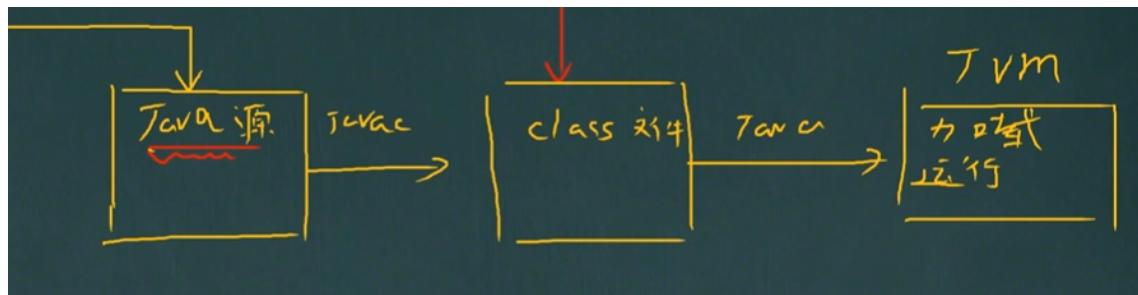
修饰注解的注解

@Retention

注解的作用范围：SOURCE, CLASS, RUNTIME

- RetentionPolicy.SOURCE：编译器使用后，直接丢弃策略的注释
- RetentionPolicy.CLASS：编译器将注解记录在class文件中，运行java程序时，jvm不会保留注释

- RetentionPolicy.RUNTIME：编译器将注解记录在class文件中，运行java程序时，jvm会保留注释，程序通过反射获取该注解



@Target

表示注解能修饰哪些元素：方法、类型...例如override只能在method

```
public enum ElementType {
    TYPE,
    FIELD,
    METHOD,
    PARAMETER,
    CONSTRUCTOR,
    LOCAL_VARIABLE,
    PACKAGE,
    TYPE_PARAMETER,
    TYPE_USE
}
```

@Documented

在形成帮助文档时保留注解

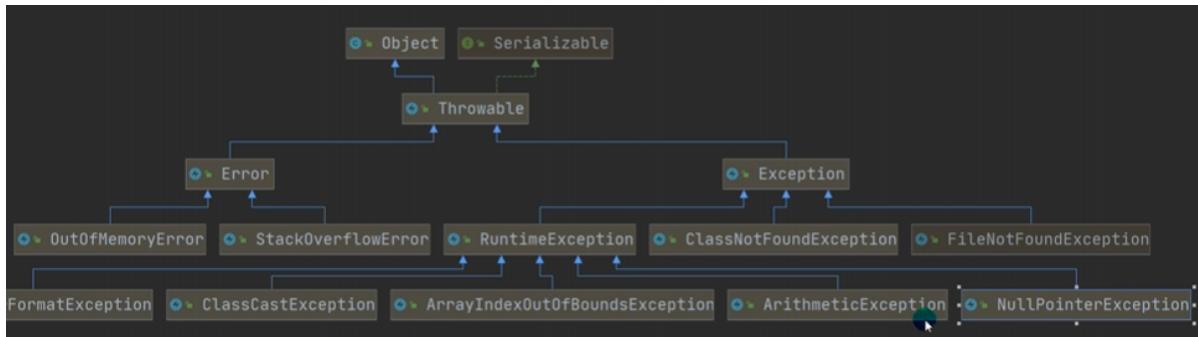
@Inherited

使被修饰的Annotation类可以有继承性

CH10 异常(Exception)

分类

1. Error: JVM无法解决的严重问题，如VM内存错误、资源耗尽(栈溢出StackOverflowError、OOM(out of Memory))
2. Exception: 因编程错误或偶然的外在因素导致的一般性问题；又分为两大类：**运行时**异常 (Runtime Error)、**编译时**异常，分别是javac编译过程的异常和java运行过程的异常
 - 编译器检查不出，运行时异常可以不做处理，很普遍
 - 编译时异常，编译器必须处置的异常



常见异常

编译异常

一般与数据库、文件相关

运行异常

`NullPointerException`

空指针异常

`ArithmaticException`

算术异常

`ArrayIndexOutOfBoundsException`

数组越界异常

`ClassCastException`

类型转换异常

`NumberFormatException`

数字格式不正确异常

异常处理方式

二选一

`try-catch-finally`

```

try{

} catch(Exception e){
// 若有异常才执行
} catch(){

} finally{
// 不管有无异常都执行
// 通常释放资源代码放在finally里
}

```

细节

1. 如果发生异常，则**异常后面的try代码都不会执行**，直接进入catch块，执行完catch执行catch后的代码块
2. 如果异常没有发生，则顺序执行try的代码块，不会进入catch
3. 如果希望发不发生异常都执行，则放在**finally**
4. 可以有多个catch语句处理多个异常，**父类异常在前，子类异常在后**
5. 不使用catch，try-finally相当于没有捕获异常，执行完finally程序直接崩溃
6. catch到异常后，后面的代码正常运行

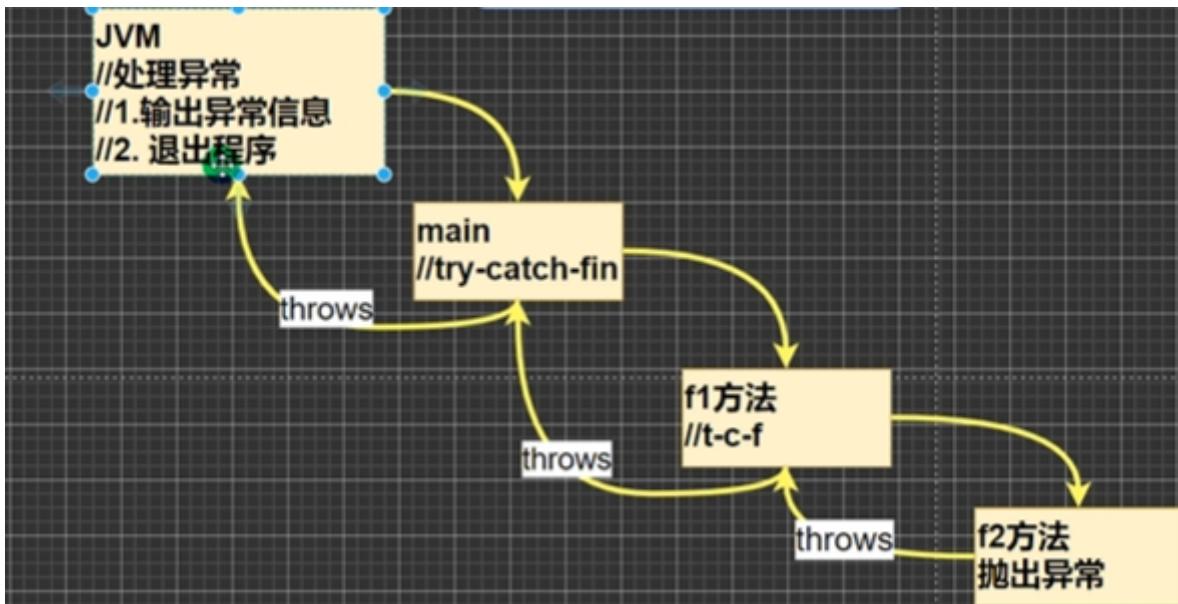
```
public class Exception02 {  
    public static int method() {  
        int i = 1;  
        try {  
            i++; //i = 2  
            String[] names = new String[3];  
            if (names[1].equals("tom")) { //空指针  
                System.out.println(names[1]);  
            } else {  
                names[3] = "hspedu";  
            }  
            return 1;  
        } catch (ArrayIndexOutOfBoundsException e) {  
            return 2;  
        } catch (NullPointerException e) {  
            return ++i; //i = 3  
        } finally { //必须执行  
            return ++i; //i = 4  
        }  
    }  
    public static void main(String[] args) {  
        System.out.println(method());  
    }  
}
```

finally优先级很高，此处++i会执行但不会马上返回，到finally才返回

throws

甩锅给上一级在（最高级是JVM）

- 如果程序显示地写t-c-f，则默认会有throws给JVM



```

public static void main(String[] args) {
    Exception04 exception04 = new Exception04();
    // 因为f1()抛出的是编译异常（区别于运行异常，如果抛出运行异常则不会报错），因此在调用处要显式的处理异常
    // 不使用try-catch或者throws会编译异常，直接报错
    try {
        exception04.f1();
    } catch (FileNotFoundException e) {
        System.out.println("输出异常啦");
    }
}
public void f1() throws FileNotFoundException, NullPointerException {
    // 也可以是throws 父类 exception
    FileInputStream fileInputStream = new FileInputStream("a.txt");
}

```

细节

1. 编译异常必须处理
2. 子类重写父类方法时，对抛出异常要么与父类一致，要么是父类抛出异常的子类
3. 不要既throws又try-catch

自定义异常

出现的错误没有实现Throwable

- 一般继承运行时异常RuntimeException，使用其默认处理机制

```

public class Exception05 {
    public static void main(String[] args) {
        int age = 123;
        if (!(age >= 12 && age <= 20)){
            throw new AgeException("年龄需要在12到20");
        }
        System.out.println("范围正确");
    }
}

```

```

class AgeException extends RuntimeException{
    public AgeException(String message) {
        super(message);
    }
}

```

throw

throw是制造/抛出异常（实例化对象），throws是处理异常

	意义	位置	后面跟的东西
throws	异常处理的一种方式	方法声明处	异常类型
throw	手动生成异常对象的关键字	方法体中	异常对象

CH11 常用类

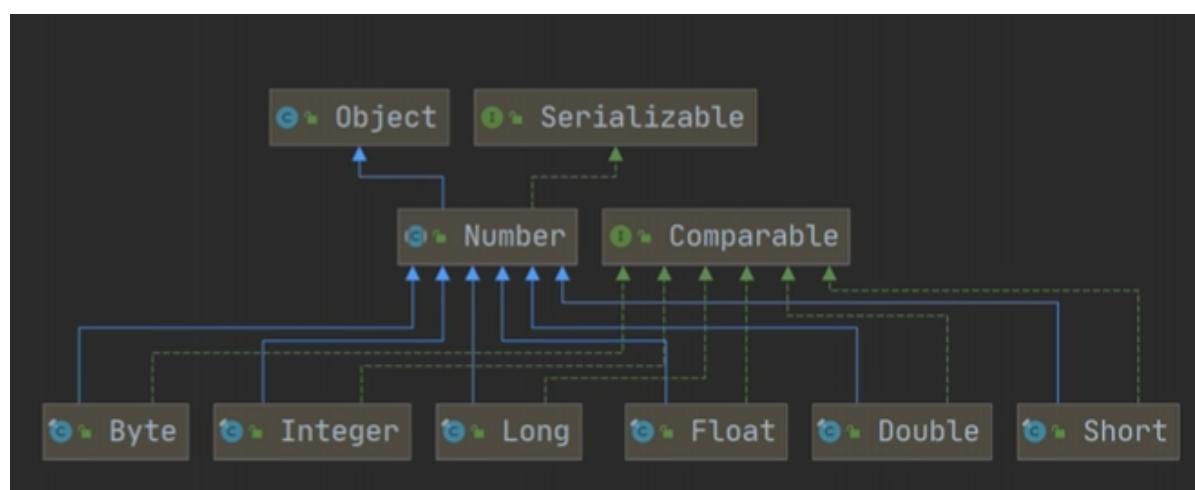
包装类(Wrapper)

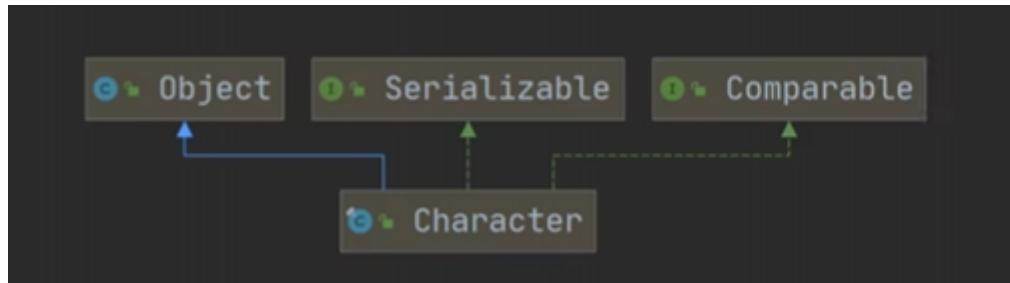
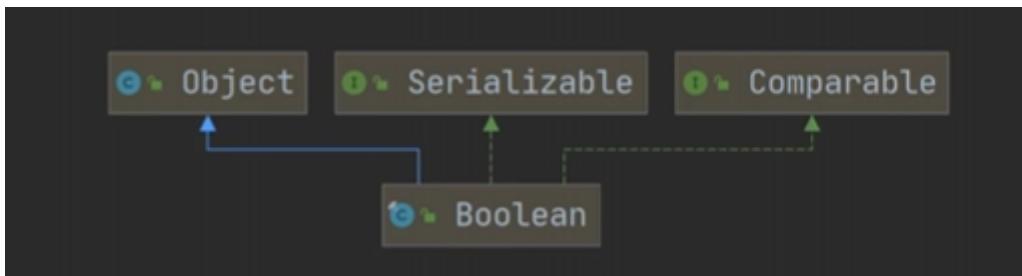
八种基本数据类型相应的应用类型——包装类

```

boolean - Boolean
char - Character
byte - Byte
short - Short
int - Integer
long - Long
float - Float
double - Double

```





装箱和拆箱

装箱：基本类型->包装类型；拆箱：包装类型->基本类型

```

// jdk5以前：手动装箱
Integer i = new Integer(100);
Integer i2 = Integer.valueOf(100);
// 手动拆箱
int j = i.intValue();
// jdk5以后：自动装箱
// 底层使用的是i3 = Integer.valueOf(100);
Integer i3 = 100;
// 自动拆箱
// 底层使用的是j2 = Integer.intValue(i3);
int j2 = i3;

```

**Object obj1 = true? new Integer(1) : new Double(2.0); //三元运算符【是一个整体】一真
大师
System.out.println(obj1); // 什么？1.0@**

三元运算符是一个整体，运算时int自动转为double，因此输出1.0

与String互转

```

// 包装类 -> String
Integer i = 100;
// 可以直接使用int对象，会自动转换
String s1 = i + "";
String s2 = i.toString();
String s3 = String.valueOf(i);
String s4 = "123";
// String -> 包装类
Integer i2 = Integer.parseInt(s4);
Integer i3 = new Integer(s4);

```

常用方法

```
System.out.println(Integer.MIN_VALUE); //返回最小值  
System.out.println(Integer.MAX_VALUE); //返回最大值  
  
System.out.println(Character.isDigit('a'));//判断是不是数字  
System.out.println(Character.isLetter('a'));//判断是不是字母  
System.out.println(Character.isUpperCase('a'));//判断是不是大写  
System.out.println(Character.isLowerCase('a'));//判断是不是小写  
  
System.out.println(Character.isWhitespace('a'));//判断是不是空格  
System.out.println(Character.toUpperCase('a'));//转成大写  
System.out.println(Character.toLowerCase('A'));//转成小写
```

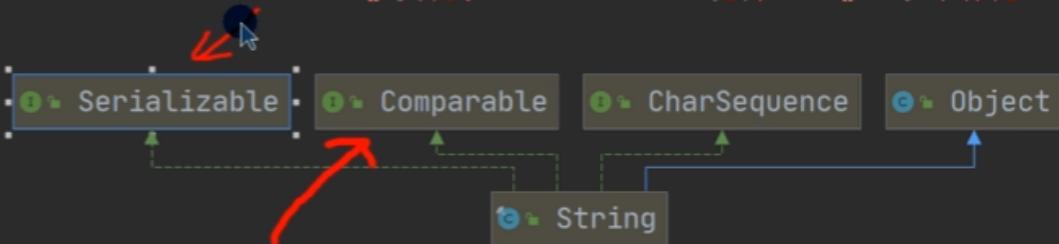
```
public void method1() {  
    Integer i = new Integer(1);  
    Integer j = new Integer(1);  
    System.out.println(i == j); //False  
    //所以，这里主要是看范围 -128 ~ 127 就是直接返回  
    Integer m = 1; //底层 Integer.valueOf(1); -> 阅读源码  
    Integer n = 1; //底层 Integer.valueOf(1);  
    System.out.println(m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

此题得关注源代码：low是-128 high是127；若在此范围则直接返回；若超出则创建新对象

```
public static Integer valueOf(int i) {  
    if (i ≥ IntegerCache.low && i ≤ IntegerCache.high)  
        return IntegerCache.cache[i + (-IntegerCache.low)]  
    return new Integer(i);  
}
```

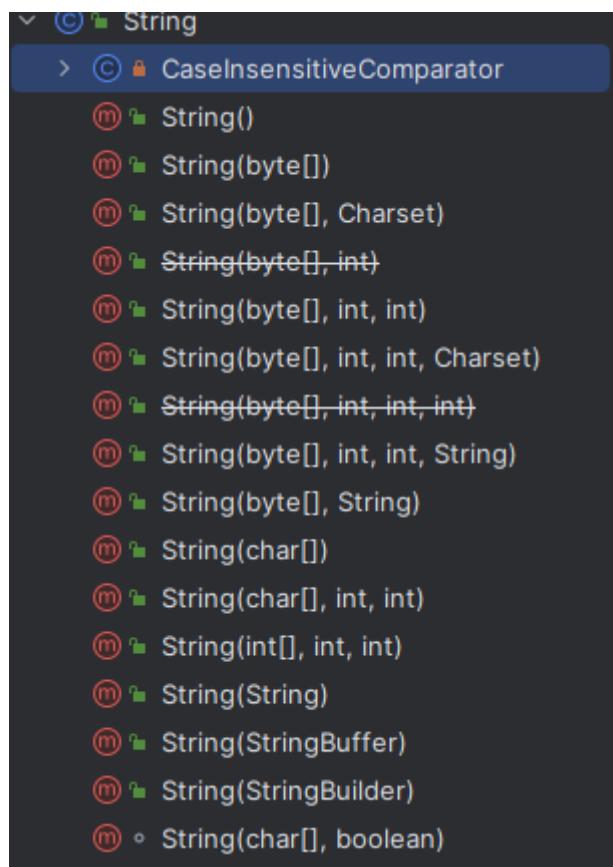
String

String 实现了Serializable，说明String 可以串行化



String实现了Comparable接口，说明String对象可以比较

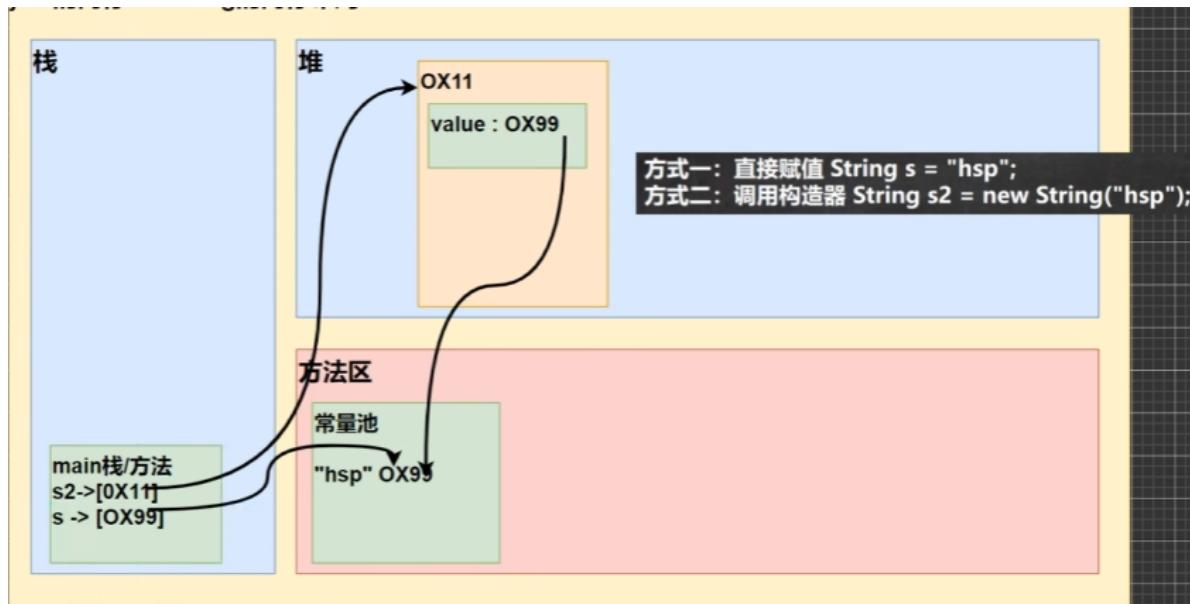
1. 双引号括起的字符序列是**字符串常量**, 例如"TES"
2. String使用的是**Unicode编码**, 汉字和字母都是**2个字节**
3. 构造器很多 (**重载**)



4. 实现了**Serializable**接口(串行化, 可以在网络传输); **Comparable**(比较String对象大小)
5. String是**final类**, 不能被其它类继承, 代表**不可变的字符序列**, 每次更新都需要重新开辟空间
6. string有属性**private final char value[]**, 用于存放字符串内容, **赋值后地址就不可被修改**; **value不可被修改, 但字符串内容可以修改**

初始化

1. 直接=: 看方法区的常量池有无对应字符串常量, 有则直接在栈中创建变量**指向此常量池的空间地址**, 没有则在常量池先创建再指
2. 构造器: **先在堆中创建空间**, 里面value值指向了常量池的字符串空间, 如果没有则创建; 如果有则value指向, **但String对象最终指向的是堆中的空间地址**



```

String a = "hsp"; //a 指向 常量池的 "hsp"
String b = new String("hsp"); //b 指向堆中对象
System.out.println(a.equals(b)); //T
System.out.println(a==b); //F
• System.out.println(a==b.intern()); //T //intern方法自己先查看API
System.out.println(b==b.intern()); //F

```

知识点:

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

老韩解读: (1) `b.intern()` 方法最终返回的是常量池的地址 (对象) .

因为常量池已经有"hsp"，因此`b.intern()`返回指向"hsp"的常量池地址

```

String s1 = "ss";
String s2 = "abcss";
// 出现常量拼接，实际使用的是构造器，因此指向的是堆
String s3 = "abc" + ss;
boolean i = s3.intern() == s2; // s3指向堆, s3.intern指向常量池

```

常用方法

`indexOf`

`lastIndexOf`

`isEmpty`

`subString 左开右闭`

`equals`

charAt

toCharArray

comcat 连接两个字符串

format

replace 返回替换过的

split 分隔正则表达式

compareTo

compareToIgnoreCase 忽略大小写

format 格式化字符串

类似于C语言风格

```
String s = "G2";
String s2 = new String("LEC");
int grade = 1;
char g = 2;
// %s %c(字符) %d(整数) %.2f 均为占位符
// 占位符用后面的变量来替换
String formatString = "战队名: %s , 赛区: %s, 成绩: %d %s";
String s3 = String.format(formatString, s, s2, grade, g);
System.out.println(s3);
```

StringBuffer

1. StringBuffer直接父类是AbstractStringBuilder，实现了Serializable类，可网络通信；本体是一个容器
2. char[] value不被final修饰，存放字符串变量，里面的值可以更改，每次更新实际上只更新内容，不更新地址（不用每次都创建对象），效率较高，直接放在堆中

StringBuffer的方法更多

与String转换

```
String str = "hello";
// String -> StringBuffer
// append或者使用构造器
StringBuffer stringBuffer = new StringBuffer(str);
stringBuffer.append(str);
// StringBuffer -> String
// toString或者使用构造器
StringBuffer stringBuffer1 = new StringBuffer();
String str2 = stringBuffer1.toString();
String s = new String(stringBuffer1);
```

常用方法

`append` 增,

`delete` 删

`replace` 改, 左闭右开

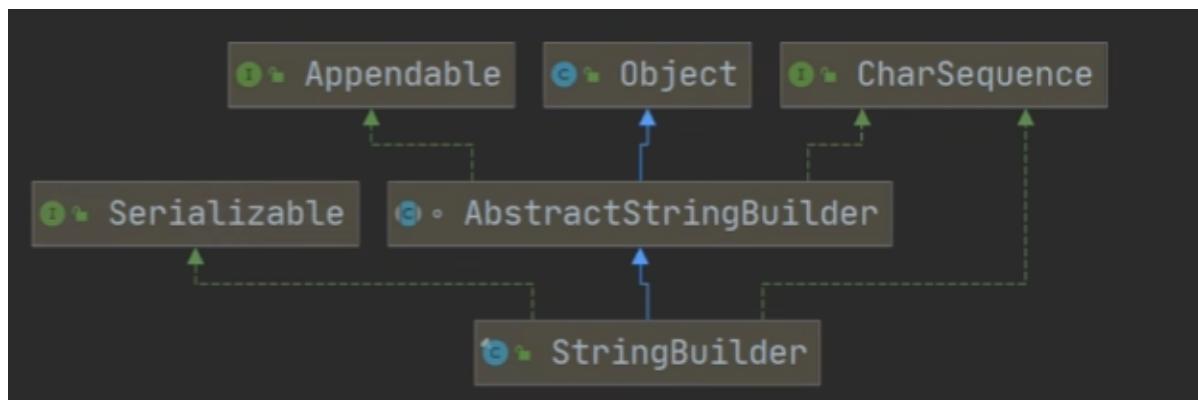
`indexOf` 查

`insert` 插, 在指定索引处插入 (扩容)

`length` 长度

StringBuilder

与StringBuffer一样是可变字符序列, 对象字符序列仍然是父类的AbstractStringBuilder, 因此**字符串地址是在堆中的**



1. 作为`StringBuffer`的简易代餐, 速度更快, **有`StringBuffer`兼容的API但不保证同步**, 并且**线程不安全**
2. 用于在字符串缓冲区被单个使用时
3. `StringBuilder`方法没有做互斥处理, 即**没有`synchronized`关键字**, 因此在单线程下使用

比较

1. `StringBuilder`: 可变字符序列, 效率最高、线程不安全
2. `StringBuffer`: 可变字符序列, 效率高、线程安全
3. `String`: 不可变字符序列, 效率低, 复用率高
4. 如果要对字符串**大量修改**, 不要使用`String`

Math

全静态

- `abs`: 绝对值
- `pow`: 求幂
- `ceil`: 向上取整
- `floor`: 向下取整
- `round`: 四舍五入

- sqrt: 开平方
- random: 随机数
- max: 最大值
- min: 最小值

Arrays

com.util.Arrays

专门用于操作数组

- `toString()`: 不用再fori这样输出了!
- `sort()`: 升序排序
- `binarySearch()`: 二分搜索查找指定的值: 数组必须提前排序号
- `copyOf(arr, arr.length)`: 数组元素的复制, `arr.length`是新数组的长度
- `fill()`: 数组元素填充
- `equals`: 比较两个数组元素内容是否完全一致
- `asList`: 将一组值转换成list

```
Integer i1[] = {2,23,134,12,4};
// 实现了Comparator的匿名内部类, 实现了要求的compare方法
// sort底层会调用compare匿名内部类, 这就是接口编程和动态绑定的好处
Arrays.sort(i1, new Comparator() {
    @Override
    public int compare(Object o1, Object o2) {
        Integer i1 = (Integer) o1;
        Integer i2 = (Integer) o2;
        return i2 - i1;
    }
});
System.out.println(Arrays.toString(i1));
```

```
// 使用sort模拟compare接口匿名内部类的实现过程
public class ArrayExercise {
    public static void main(String[] args) {
        Book[] books = new Book[5];
        books[0] = new Book("红楼梦", 200);
        books[1] = new Book("厕纸", 60);
        books[2] = new Book("RAP", 123);
        books[3] = new Book("lol", 333);
        books[4] = new Book("sad", 213);
        // 匿名内部类
        sort(books, new Comparator() {
            @Override
            public int compare(Object o1, Object o2) {
                Book book1 = (Book) o1;
                Book book2 = (Book) o2;
                return book1.getPrice() - book2.getPrice();
            }
        });
    }
}
```

```

        return book1.getName().length() - book2.getName().length();
    }
}
System.out.println(Arrays.toString(books));
}

public static void sort(Book[] books, Comparator comparator){
    Book temp;
    for (int i = 0; i < books.length - 1; i++) {
        for (int j = 0; j < books.length - i - 1; j++) {
            if (comparator.compare(books[j], books[j+1]) > 0){
                temp = books[j];
                books[j] = books[j+1];
                books[j+1] = temp;
            }
        }
    }
}

class Book{
    private String name;
    private int price;

    public String getName() {
        return name;
    }

    public int getPrice() {
        return price;
    }

    public Book(String name, int price) {
        this.name = name;
        this.price = price;
    }

    @Override
    public String toString() {
        return "Book{" +
            "name='" + name + '\'' +
            ", price=" + price +
            '}';
    }
}

```

System

- exit: 退出当前程序
- arraycopy: 复制数组元素，实际上会覆盖
- currentTimeMillis: 返回当前时间与19700101的毫秒数

```
public static native void arraycopy(Object src, int srcPos,
                                    Object dest, int destPos,
                                    int length)

// src: 源数组; srcPos: 从源数组的哪个索引位置开始拷贝
// des: 目标数组; destPos: 目标数组的索引
// length: 拷贝多少个
```

- gc: 运行垃圾回收机制

```
// System.exit(0);
System.out.println("MAMBA OUT");
int a1[] = {21, 22, 31, 234, 122};
int a2[] = {12, 341, 52};
System.arraycopy(a1, 0, a2, 1, 2);
System.out.println(Arrays.toString(a2));
System.out.println(System.currentTimeMillis());
```

BigInteger BigInteger

用来表示很大很大的integer和double

因为是类不是基本数据类型，所以要用对应的方法而不是直接运算

```
BigInteger bigInteger = new BigInteger("2222222222222222");
bigInteger = bigInteger.add(new BigInteger("1233"));
System.out.println(bigInteger.toString());
```

日期

第一代: Date

获取当前时间

```
// 有参数构造为指定毫秒数
public static void main(String[] args) throws ParseException {
    Date date = new Date();
    System.out.println(date);
    SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy年MM月dd日
hh:mm:ss E");
    String format = simpleDateFormat.format(date);
    System.out.println(format);
    String s2 = "2000年12月21日 10:20:30 星期一";
    // 使用指定的String格式转换，需要捕获ParseException对象
    Date parse = simpleDateFormat.parse(s2);
    System.out.println(format);
}
// out都为
//Sat May 18 12:08:41 CST 2024
//2024年05月18日 12:08:41 星期六
```

第二代：Calender

- 不是线程安全的，不能处理闰秒

```
// 是抽象类而且构造器是private  
// 通过getInstance()获取实例  
Calendar calendar = Calendar.getInstance();  
// 获取年  
System.out.println(calendar.get(Calendar.YEAR));
```

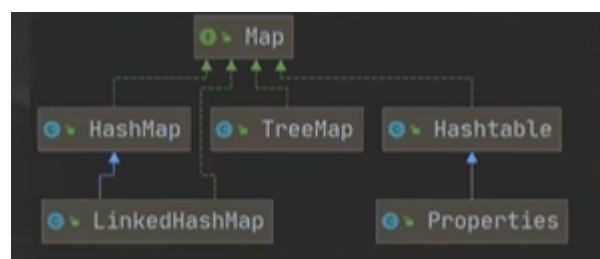
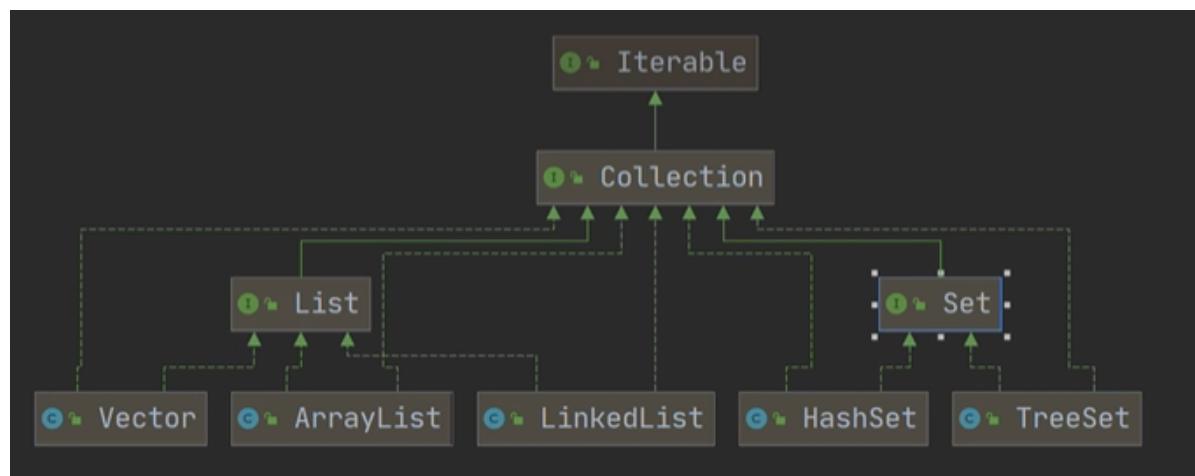
第三代：LocalDate LocalTime LocalDateTime

- jdk8之后加入

```
LocalDateTime localDateTime = LocalDateTime.now();  
// 当前时间  
System.out.println(localDateTime);  
// 年  
System.out.println(localDateTime.getYear());  
DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("yyyy年MM月dd日  
HH小数mm分钟ss秒");  
// 自定义格式化 日期  
String format = dateTimeFormatter.format(localDateTime);  
System.out.println("格式化的日期=" + format);
```

CH12 集合类

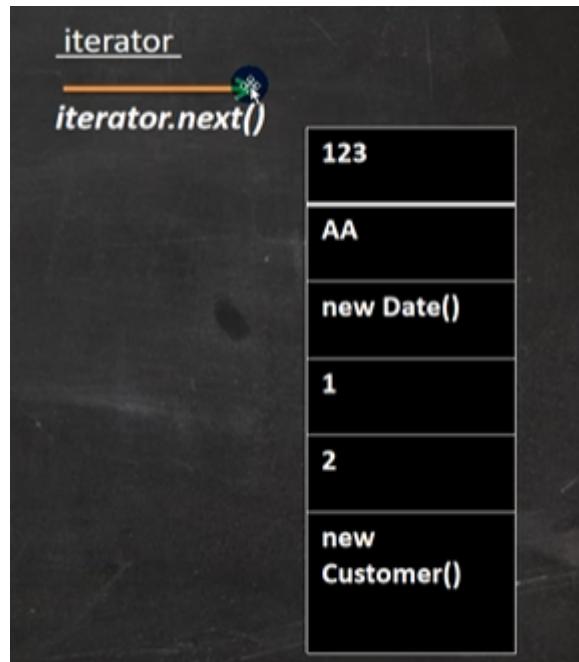
- 单列集合：Collection：List, Set
- 双列集合（键-值对）：Map



遍历

Iterator

- 位于Iterable中，是Iterable接口的成员内部接口（类）
- 有3个函数，其中next()和hasNext()比较常用，用于遍历集合，快捷键itit



增强for

快捷键l，底层代码仍然是iterator，简化版的迭代器

```
public class CollectionIterator {
    public static void main(String[] args) {
        Collection col = new ArrayList();
        col.add(new Book2("三国演义", "罗贯中", 10.2));
        col.add(new Book2("水浒传", "施耐庵", 12.2));
        col.add(new Book2("红楼梦", "曹雪芹", 13.2));
        col.add("4396");
        Iterator iterator = col.iterator();
        // 快捷键itit
        while (iterator.hasNext()) {
            Object next = iterator.next();
            System.out.println(next);
        }
        // Exception: NoSuchElementException
        iterator.next();
        // 如果再次遍历，需要重置迭代器
        iterator = col.iterator();
        for (Object book : col) {
            System.out.println("book = " + book);
        }
    }
}
class Book2{
    private String name;
```

```

private String author;
private double price;

public Book2(String name, String author, double price) {
    this.name = name;
    this.author = author;
    this.price = price;
}

@Override
public String toString() {
    return "Book{" +
        "name='" + name + '\'' +
        ", author='" + author + '\'' +
        ", price=" + price +
        '}';
}
}

```

普通for

本质是因为List有索引有get()方法，如果为Set接口则不行

```

for (int i = 0; i < list.size(); ++i)
{
    Object o = list.get(i);
}

```

Collection

常用方法

```

List list = new ArrayList();
list.add("jack");
// 此处自动装箱了把12装箱为Integer对象
// 相当于list.add(new Integer(12));
list.add(new Integer(12));
list.remove(0);
System.out.println(list);
System.out.println("contains() = " + list.contains("jack"));
System.out.println("size() = " + list.size());
System.out.println("isEmpty() = " + list.isEmpty());
list.clear();
System.out.println("clear() = " + list);
List list2 = new ArrayList();
list2.add("好好");
list2.add("azur");
list.addAll(list2);
System.out.println("addAll() = " + list);
list.removeAll(list2);
System.out.println("removeAll() = " + list);

```

- 1) add:添加单个元素
- 2) remove:删除指定元素
- 3) contains:查找元素是否存在
- 4) size:获取元素个数
- 5) isEmpty:判断是否为空
- 6) clear:清空
- 7) addAll:添加多个元素
- 8) containsAll:查找多个元素是否都存在
- 9) removeAll: 删除多个元素

Collections

用于操作集合类

常用方法

- 1) reverse(List): 反转 List 中元素的顺序
- 2) shuffle(List): 对 List 集合元素进行随机排序
- 3) sort(List): 根据元素的自然顺序对指定 List 集合元素按升序排序
- 4) sort(List, Comparator): 根据指定的 Comparator 产生的顺序对 List 集合元素进行排序
- 5) swap(List, int, int): 将指定 list 集合中的 i 处元素和 j 处元素进行交换
- 6) 应用案例演示 Collections.java

- add
- sort(Comparator)
- reverse
- max

可以自定义Comparator

```
collections.max(list, new Comparator() {  
    @Override  
    public int compare(Object o1, Object o2){  
        return ((String)o1).length() - ((String)o2).length();  
    }  
});
```

- min
- copy
- swap
- copy
- replaceAll

```

List list = new ArrayList();
list.add("hihi");
list.add("hello");
list.add("hello2");
list.add("hello23");

Collections.reverse(list);
Collections.sort(list, new Comparator() {
    @Override
    public int compare(Object o1, Object o2){
        return ((String)o1).length() - ((String)o2).length();
    }
});

```

List

元素有序，且有自己的索引，可以用get()方法取对应索引的元素

ArrayList

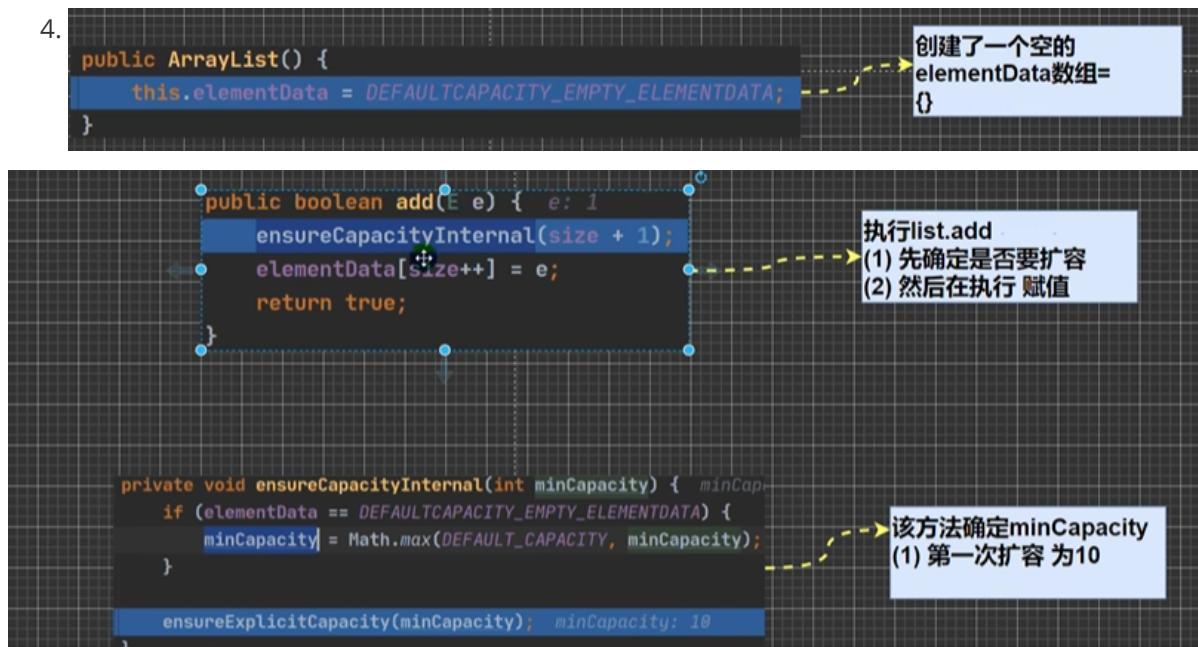
细节

1. 可以添加null为元素，且可以有多个
2. 底层数据存储为数组实现的
3. 线程不安全，源代码没有加关键字synchronized，但效率高
4. 基本等于Vector，多线程情况下不建议用ArrayList

源码

维护了一个Object型数组elementData， transient Object[] elementData;(transient表示属性不会被序列化)

1. 使用默认构造器，elementData的容量为0
2. 第一次添加，扩容后elementData为10，要再次扩容，扩容为1.5倍
3. 如果使用指定大小构造器，初始容量为指定大小，若再次扩容则为1.5倍



```

private void ensureExplicitCapacity(int minCapacity) {
    modCount++;

    // overflow-conscious code
    if (minCapacity - elementData.length > 0) minCapacity = grow(minCapacity);
}

private void grow(int minCapacity) { minCapacity: 10
    // overflow-conscious code
    int oldCapacity = elementData.length; oldCapacity (slot
    int newCapacity = oldCapacity + (oldCapacity >> 1); newC
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity); minCapacit
    // minCapacity is usually close to size, so this is a wi
    elementData = Arrays.copyOf(elementData, newCapacity);
}

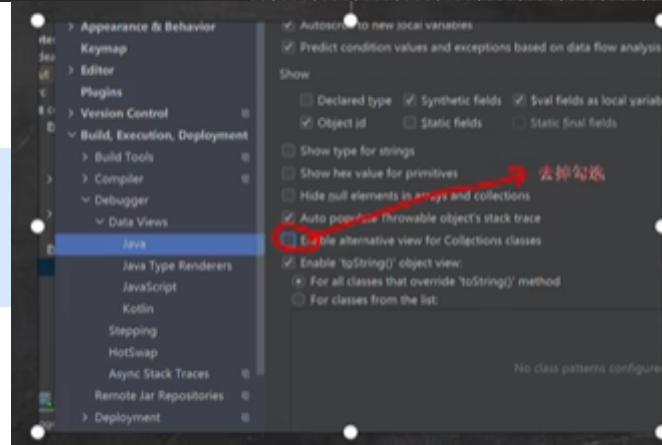
```

(1) modCount++ 记录集合被修改的次数
(2) 如果elementData的大小不够,就调用grow()去扩容

(1) 真的扩容
(2) 使用扩容机制来确定要扩容到多大
(3) 第一次newCapacity = 10
(4)

韩顺平
教育

- (4) 第二次及其以后, 按照1.5倍扩容
- (5) 扩容使用的是Arrays.copyOf()



常用方法

- add(int index, Object) : 在指定索引处添加
- indexOf(int index, Object o): 返回o首次出现的位置
- lastIndexOf(int index, Object): 返回最后出现的位置
- remove(int index): 移除指定索引元素, 并返回此元素
- set(int index, Object o): 将索引处元素替换为o
- subList(int i, int j): 左闭右开

```

ArrayList list = new ArrayList();
list.add("jack");
list.add("tom");
list.add("mary");
list.add(1, "jacklove");
System.out.println(list);
list.clear();
for (int i = 1; i <= 10; i++) {
    list.add(i);
}
list.add(2, "faker");
Object o = list.get(4);
list.remove(5);

```

```

list.set(6, "dog");
Iterator iterator = list.iterator();
// while
while (iterator.hasNext()) {
    Object next = iterator.next();
    System.out.println(next);
}
// 普通for
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
// 增强for
for (Object object : list) {
    System.out.println(object);
}

```

冒泡排序

```

public static void main(String[] args) {
    // ArrayList
    ArrayList col = new ArrayList();
    col.add(new Book2("三国演义", "罗贯中", 10.2));
    col.add(new Book2("水浒传", "施耐庵", 19.2));
    col.add(new Book2("红楼梦", "曹雪芹", 16.2));
    col.add(new Book2("厕纸", "Letme", 2));
    bubbleSort(col, new Comparator() {
        @Override
        public int compare(Object o1, Object o2) {
            return ((Book2)o1).getPrice() - ((Book2)o2).getPrice();
        }
    });
    System.out.println(col);
    // LinkedList
    LinkedList linkedList = new LinkedList();
    linkedList.add(new Book2("三国演义", "罗贯中", 10.2));
    linkedList.add(new Book2("水浒传", "施耐庵", 19.2));
    linkedList.add(new Book2("红楼梦", "曹雪芹", 16.2));
    linkedList.add(new Book2("厕纸", "Letme", 2));
    bubbleSort(linkedList, new Comparator() {
        @Override
        public int compare(Object o1, Object o2) {
            return ((Book2)o1).getPrice() - ((Book2)o2).getPrice();
        }
    });
    System.out.println(linkedList);
    // Vector实现
    Vector vector = new Vector();
    vector.add(new Book2("三国演义", "罗贯中", 10.2));
    vector.add(new Book2("水浒传", "施耐庵", 19.2));
    vector.add(new Book2("红楼梦", "曹雪芹", 16.2));
    vector.add(new Book2("厕纸", "Letme", 2));
    bubbleSort(vector, new Comparator() {
        @Override
        public int compare(Object o1, Object o2) {
            return ((Book2)o1).getPrice() - ((Book2)o2).getPrice();
        }
    });
}
```

```

        }
    });
    System.out.println(vector);

}

public static void bubbleSort(List list, Comparator comparator){
    Object temp;
    for (int i = 0; i < list.size() - 1; i++) {
        for (int j = 0; j < list.size() - i - 1; j++) {
            if ((comparator.compare(list.get(j), list.get(j+1)) > 0){
                temp = list.get(j+1);
                list.set(j+1, list.get(j));
                list.set(j, temp);
            }
        }
    }
}

```

Vector

1. 底层也是object数组， protected Object[] elementData;
2. 线程同步即**线程安全**，其操作方法都带有**synchronized**
3. 线程同步安全时，考虑用Vector
4. 底层与ArrayList一样都是**数组**

扩容方式：无参默认10，满后按2倍扩

● Vector和ArrayList的比较

	底层结构	版本	线程安全（同步）	效率	扩容倍数
ArrayList	可变数组	jdk1.2	不安全，效率高	如果有参构造1.5倍 如果是无参 1.第一次10 2.从第二次开始按1.5倍	
Vector	可变数组	jdk1.0	安全，效率不高	如果是无参， 默认10 ，满后，就按2倍扩容 如果指定大小，则每次直接按2倍扩	

LinkedList

- 双向链表和双端队列
- 可以添加任意元素，包括null
- 线程不安全，没有实现同步

Node

- 可以理解为2个指针(next + prev) +1个属性值
- 比较费解的是next和prev同样也是Node

```
class Node {  
    public Object item;  
    public Node next;  
    public Node prev;  
  
    public Node(Object item) {  
        this.item = item;  
    }  
}
```

应用

```
public class LinkedList_ {  
    public static void main(String[] args) {  
        Node jack = new Node("jack");  
        Node tom = new Node("tom");  
        Node hihi = new Node("hihi");  
  
        jack.next = tom;  
        tom.next = hihi;  
        tom.pre = jack;  
        hihi.pre = tom;  
        Node last = hihi;  
  
        while (true){  
            if (last == null){  
                break;  
            }  
            System.out.println(last);  
            last = last.pre;  
        }  
    }  
}  
  
class Node {  
    public Object item;  
    public Node next;  
    public Node pre;  
  
    public Node(Object item) {  
        this.item = item;  
    }  
  
    @Override  
    public String toString() {  
        return "Node{" +  
               "item=" + item +  
               '}';  
    }  
}
```

```
}
```

CRUD

```
// LinkedList linkedList = new LinkedList();相当于
// 1. public LinkedList() {}仅作初始化
// 2. 此时linkedlist.first = null, linkedlist.last = null
LinkedList linkedList = new LinkedList();
// 3. 执行add
//     public boolean add(E e) {
//         linkLast(e);
//         return true;
//     }
// 4. 将新的结点加入到双向链表的最后
// 这段代码2个功能：加入新的结点；更新LinkedList的first和last
//     void linkLast(E e) {
//         final Node<E> l = last;
//         final Node<E> newNode = new Node<>(l, e, null);
//         last = newNode;
//         if (l == null)
//             first = newNode;
//         else
//             l.next = newNode;
//         size++;
//         modCount++;
//     }
linkedList.add(1);
linkedList.add(2);
// return removeFirst();
//     public E removeFirst() {
// final Node<E> f = first这一步赋值很像冒泡排序的int temp = a[j]
//         final Node<E> f = first;
//         if (f == null)
//             throw new NoSuchElementException();
//         return unlinkFirst(f);
//     }
// 5.删除第一个结点
// 创建变量存储第二个结点地址，第一个指针的值=null是帮忙清除缓存
// 然后清除第一个结点，再更新LinkedList的first和把第二个结点的prev置空
//     private E unlinkFirst(Node<E> f) {
//         // assert f == first && f != null;
//         final E element = f.item;
//         final Node<E> next = f.next;
//         f.item = null;
//         f.next = null; // help GC
//         first = next;
//         if (next == null)
//             last = null;
//         else
//             next.prev = null;
//         size--;
//         modCount++;
//         return element;
//     }
```

```
linkedList.remove();
System.out.println(linkedList);
```

比较

	底层结构	增删的效率	改查的效率
ArrayList	可变数组	较低 数组扩容	较高
LinkedList	双向链表	较高，通过链表追加.	较低

- 改查多，用ArrayList（数组）
- 增删多，用LinkedList（链表）
- 程序80-90%都是查询，大部分情况选择ArrayList
- 两者都线程不安全

Set

1. 无序(添加和取出的顺序不一致)，但顺序是固定的，没有索引
2. 不允许重复元素/对象，最多只能包含一个null
3. 实现了Collection，有Iterator以及增强for用于遍历
4. 不能使用索引的方法来获取

HashSet

1. 本质是一个HashMap(数组+链表)

```
@Contract(pure = true)
public HashSet() { map = new HashMap(); }
```

2. 添加一个元素时，会得到hash值，转换成索引值
3. 找到存储表table，看这个索引值对应的元素是否已经存放有元素：如果没有，则直接加入；如果有，则调用equals比较，如果相同放弃添加；如果不同，则添加到最后
4. java8中，如果一条链表的元素个数超过TREEIFY_THRESHOLD(默认是8)，并且table的大小
 $\geq \text{MIN_TREEIFY_CAPACITY}$ (默认64)，会进化成红黑树

关于table (数组+链表) 解释：可用如下解释

```
// 2. Node[16] 容器插入操作实现
Node02[] table = new Node02[16];  table: Node02[16]@501
System.out.println(table);
// 3. 创建结点
Node02 john = new Node02(item: "john", next: null);  john: Node02@502
table[2] = john;
Node02 jack = new Node02(item: "jack", next: null);  jack: Node02@503
john.next = jack;  john: Node02@502
Node02 rose = new Node02(item: "rose", next: null);  rose: Node02@504
jack.next = rose;  jack: Node02@503  rose: Node02@504
Node02 lucy = new Node02(item: "lucy", next: null);  lucy: Node02@505
table[3] = lucy;  lucy: Node02@505
System.out.println("table = " + table);  table: Node02[16]@501
}

}

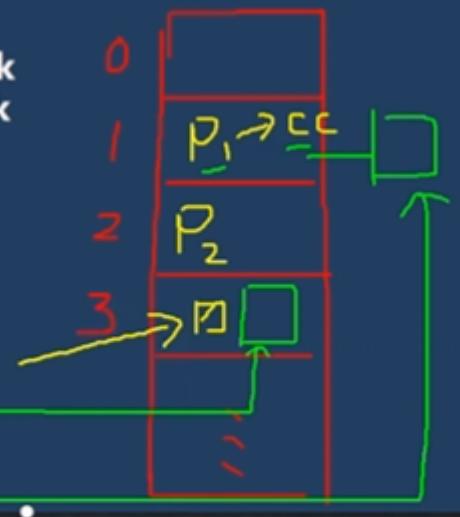
12 usages
class Node02 {
    3 usages
    Node02 next;
    1 usage
    Object item;

    4 usages
    public Node02(Object item, Node02 next) {
        this.next = next;
        this.item = item;
    }
}

tail x HashSetStructure x
Console C | D | M | S | L | U | P | O | F | : 
Evaluate expression (Enter) or add a watch (Ctrl+Shift+Enter)
ture (con)  ⚡ args = {String[0]@500} []
  ↴ table = {Node02[16]@501}
      Not showing null elements
      ↴ 2 = {Node02@502}
          ↴ next = {Node02@503}
              ↴ next = {Node02@504}
                  ↴ next = null
                  > ↴ item = "rose"
                  > ↴ item = "jack"
                  > ↴ item = "john"
          > ↴ 3 = {Node02@505}
      > ↴ john = {Node02@502}
      > ↴ jack = {Node02@503}
      > ↴ rose = {Node02@504}
```

源码

```
HashSet set = new HashSet(); //ok
Person p1 = new Person(1001,"AA"); //ok
Person p2 = new Person(1002,"BB"); //ok
set.add(p1); //ok
set.add(p2); //ok
p1.name = "CC";
set.remove(p1);
System.out.println(set); ②
set.add(new Person(1001,"CC"));
System.out.println(set); ③
set.add(new Person(1001,"AA"));
System.out.println(set); ④
```



1. p1.name更改后，p1的hash值发生改变，`hashSet.remove(p1);`删除不了，因为原先p1所在的哈希表的位置哈希值是由"1001""AA"确定的（哈希表中的位置已经固定），现在删除p1的哈希值是由"1001""CC"确定的
2. `hashSet.add(newPerson02("1001","CC"));`能够加进去，理由同上
3. `hashSet.add(newPerson02("1001","AA"));`能够加进去，会加在第一个p1的next里，即处于同一数组，因为他们的哈希值相同所以位置确定，并且key值不同所以成功加入

putVal

```
// 形参: hash: 要存入的hash值, key: 要存入的key值, value: 要存入的value值
final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
               boolean evict) {
    // 初始化临时变量: tab: 当前链表(结点)数组; p: 指向的结点
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    // 初始化, 链表为0时创建新列表
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    // (1) 链表对应索引为空
    // (1.1) 若该随机索引处没有数值存储, 则直接存进去
    // /此处tab[i = (n - 1) & hash]返回的是一个介于0到n-1的值, 这也是哈希表中随机插入的体现
    // /随机性体现在位与hash值, hash值虽然固定但又有些无厘头, 并不是传统的按顺序插入/
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    // (1.2) 若该结点数组对应的随机索引值处已经有链表存在
    else {
        Node<K,V> e; K k;
        // (2) 链表对应索引非空
        // (2.1) 与链表数组的第一个索引值相等, 同时key相等, 说明为重复元素!
        // /(k = p.key) == key为直接==比较, 效率高(优先);
        // /key.equals(k)为类自己定义对的equals(动态绑定)比较, 效率低
        if (p.hash == hash &&
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        // (2.2) 是树结点?
        else if (p instanceof TreeNode)
```

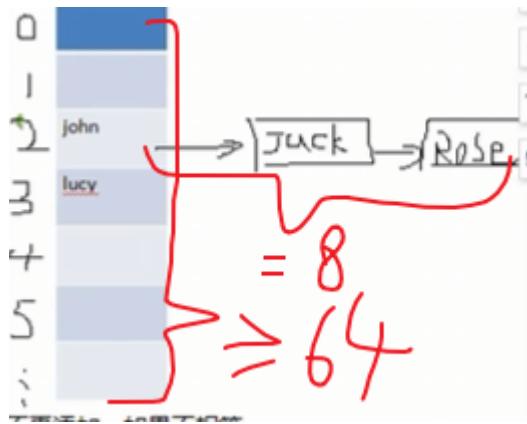
```

        e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        // (2.3) 与链表数组第一个索引值不等，开始遍历判断是否有相等的时候
    else { // 遍历开始
        for (int binCount = 0; ; ++binCount) {
            // 遍历到了最后，确实没有找到与要插入的key值相同的，即非重复元素，则插入！
            // 此时e=null,此处e = p.next即为循环启动器
            if ((e = p.next) == null) {
                p.next = newNode(hash, key, value, null);
                // 突破极限，开始树化
                if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                    treeifyBin(tab, hash);
                break;
            }
            // 确实跟数组的某一个元素的key值相等
            if (e.hash == hash &&
                ((k = e.key) == key || (key != null && key.equals(k))))
                break;
            p = e;
        }
    }
    // (3.1) e被标记过，非null说明确实是重复元素
    if (e != null) { // existing mapping for key
        v oldValue = e.value;
        if (!onlyIfAbsent || oldValue == null)
            e.value = value;
        afterNodeAccess(e);
        return oldValue;
    }
}
// (3.2) 执行到此处说明e为null，说明非重复元素
++modCount;
if (++size > threshold)
    resize();
afterNodeInsertion(evict);
return null;
}

```

treeifyBin

1. 当链表元素是达到TREEIFY_THRESHOLD(默认是8)，注意是table中的所有元素加起来而不是单个数组
2. 并且table的大小 \geq MIN_TREEIFY_CAPACITY(默认64)时，就会进行树化
3. 当同一个数组中的元素 $>=8$ (与1不同，此处要求是同一个数组，而不是任意) 时会开始扩容，一次扩2倍
4. 当 $\geq=0.75 * \text{扩容阈值}$ 时也会开始扩容



```

final void treeifyBin(Node<K,V>[] tab, int hash) {
    int n, index; Node<K,V> e;
    if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
        resize();
    else if ((e = tab[index = (n - 1) & hash]) != null) {
        TreeNode<K,V> hd = null, tl = null;
        do {
            TreeNode<K,V> p = replacementTreeNode(e, null);
            if (tl == null)
                hd = p;
            else {
                p.prev = tl;
                tl.next = p;
            }
            tl = p;
        } while ((e = e.next) != null);
        if ((tab[index] = hd) != null)
            hd.treeify(tab);
    }
}

```

resize()

```

final Node<K,V>[] resize() {
    // 初始化各种变量, oldCap: 旧容量; newCap: 新容量
    // oldThr: 旧阈值限; newThr: 新阈值
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        // 旧阈值
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        } // 旧阈值突破了, 新容量(2倍后)小于极限值, 成功扩容2倍
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else { // zero initial threshold signifies using defaults

```

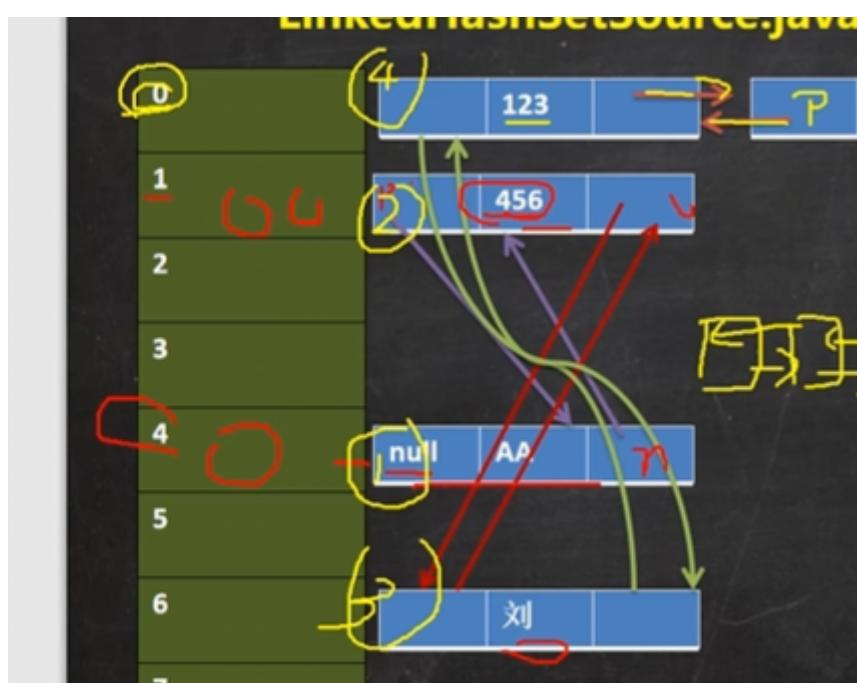
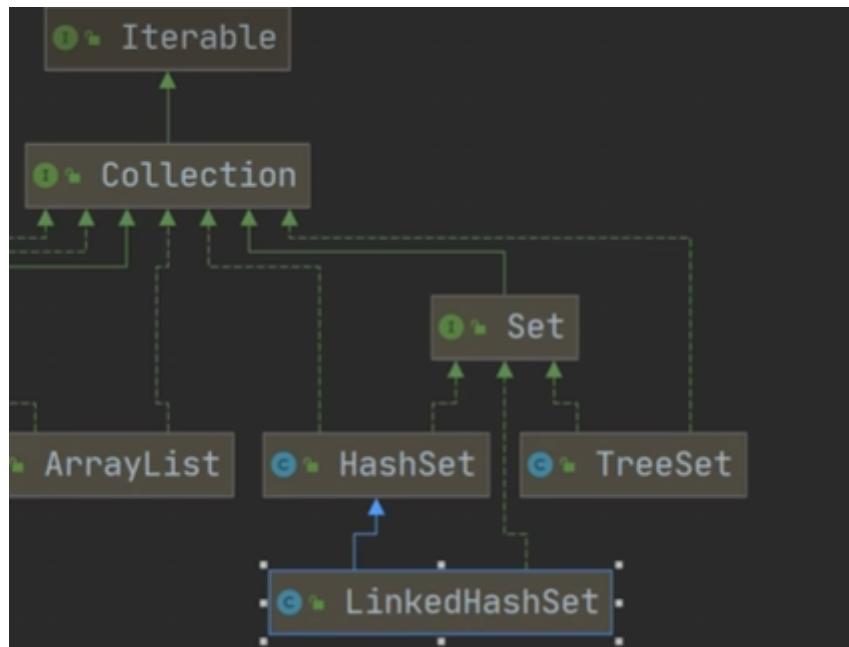
```

        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
            (int)ft : Integer.MAX_VALUE);
    }
    threshold = newThr;
    @SuppressWarnings({"rawtypes", "unchecked"})
    Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
    table = newTab;
    if (oldTab != null) {
        for (int j = 0; j < oldCap; ++j) {
            Node<K,V> e;
            if ((e = oldTab[j]) != null) {
                oldTab[j] = null;
                if (e.next == null)
                    newTab[e.hash & (newCap - 1)] = e;
                else if (e instanceof TreeNode)
                    ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                else { // preserve order
                    Node<K,V> loHead = null, loTail = null;
                    Node<K,V> hiHead = null, hiTail = null;
                    Node<K,V> next;
                    do {
                        next = e.next;
                        if ((e.hash & oldCap) == 0) {
                            if (loTail == null)
                                loHead = e;
                            else
                                loTail.next = e;
                            loTail = e;
                        }
                        else {
                            if (hiTail == null)
                                hiHead = e;
                            else
                                hiTail.next = e;
                            hiTail = e;
                        }
                    } while ((e = next) != null);
                    if (loTail != null) {
                        loTail.next = null;
                        newTab[j] = loHead;
                    }
                    if (hiTail != null) {
                        hiTail.next = null;
                        newTab[j + oldCap] = hiHead;
                    }
                }
            }
        }
    }
    return newTab;
}

```

LinkedHashSet

1. 是HashSet的子类
2. 底层是LinkedHashMap，底层维护数组+双向链表
3. 根据元素的hashCode值决定存储位置，同时使用链表维护元素次序



1个结点=1个元素+1个before+1个after+1个next

源码

1. 添加第一次时，将数组table扩容到16，存放的结点类型是LinkedHashMap@Entry，作为静态内部类继承了HashMap.Node

```
static class Entry<K, V> extends HashMap.Node<K, V> {  
    Entry<K, V> before, after;
```

2. 扩容底层仍是HashSet那套putVal
3. 不同的是添加元素时，先求hash值，再求索引，确定该元素在table表中的位置，然后将添加的元素加入到双向链表中，因为有before和after，因而变得有序

```
LinkedHashSet linkedHashSet =new LinkedHashSet();
linkedHashSet.add(456);
linkedHashSet.add(456);
linkedHashSet.add("ss");
linkedHashSet.add("aa");
System.out.println(linkedHashSet);
```

```
✓ 0 = {LinkedHashMap$Entry@546} "ss=java.lang.Object@136432db"
  > ⚡ before = {LinkedHashMap$Entry@534} "456=java.lang.Object@136432db"
  > ⚡ after = {LinkedHashMap$Entry@553} "aa=java.lang.Object@136432db"
    ⚡ hash = 3680
  > ⚡ key = "ss"
  > ⚡ value = {Object@565}
  > ⚡ next = {LinkedHashMap$Entry@553} "aa=java.lang.Object@136432db"
✓ 8 = {LinkedHashMap$Entry@534} "456=java.lang.Object@136432db"
  ⚡ before = null
  > ⚡ after = {LinkedHashMap$Entry@546} "ss=java.lang.Object@136432db"
    ⚡ hash = 456
  > ⚡ key = {Integer@564} 456
  > ⚡ value = {Object@565}
```

TreeSet

能够利用comparator的匿名内部类实现有序排列

```
TreeSet treeSet = new TreeSet(new Comparator() {
    @Override
    public int compare(Object o1, Object o2) {
        return ((String)o1).length() - ((String) o2).length();
    }
});
treeSet.add("jack");
treeSet.add("tom");
treeSet.add("fffffff");
treeSet.add("haha23");
// 注意！不会把haha23添加进去，因为compare之后==0
System.out.println(treeSet);
```

源码

```
public V put(K key, V value) {
    Entry<K,V> t = root;
    if (t == null) {
        compare(key, key); // type (and possibly null) check
```

```

        root = new Entry<>(key, value, null);
        size = 1;
        modCount++;
        return null;
    }

    int cmp;
    Entry<K,V> parent;
    // split comparator and comparable paths
    Comparator<? super K> cpr = comparator;
    if (cpr != null) {
        do {
            parent = t;
            // 动态绑定到匿名内部类
            cmp = cpr.compare(key, t.key);
            if (cmp < 0)
                t = t.left;
            else if (cmp > 0)
                // 大的那个放右边, 升序
                t = t.right;
            // 注意当cmp == 0 时直接return, 不会添加
            else
                return t.setValue(value);
        } while (t != null);
    }
    ...
}

```

比较

- HashSet: 基于 hashCode() 和 equals() 实现去重, 底层存入对象进行位运算得到 hash 值, 再找到 hash 值对应的索引, 如果索引处非空则使用 == 或者 equals 进行遍历比较
- TreeSet: 基于 Comparator: 若传入的是 Comparator 匿名对象, 则使用匿名对象实现的 compareTo 去重; 若没有创建匿名对象, 则以添加的对象实现的 Comparable 接口的 compareTo 去重

```

public class Homework05 {
    public static void main(String[] args) {
        TreeSet treeSet = new TreeSet();
        treeSet.add(new Person());
    }
}

1 usage
class Person{
}

```

会抛出类型转换异常, 原因在于 TreeSet 在进行 add 时会使用会将 key 向下转型为 Comparable 对象以便于调用 compareTo 函数进行去重, 由于 Person 类没有实现 Comparable, 因此报错

```

final int compare(Object k1, Object k2) {
    return comparator==null ? ((Comparable<? super K>)k1).compareTo((K)k2)
        : comparator.compare((K)k1, (K)k2);
}

```

正解：实现Comparable接口，并重写comapreTo方法

```

public class Homework05 {
    public static void main(String[] args) {
        TreeSet treeSet = new TreeSet();
        treeSet.add(new Person());
        treeSet.add(new Person());
        treeSet.add(new Person());
    }
}

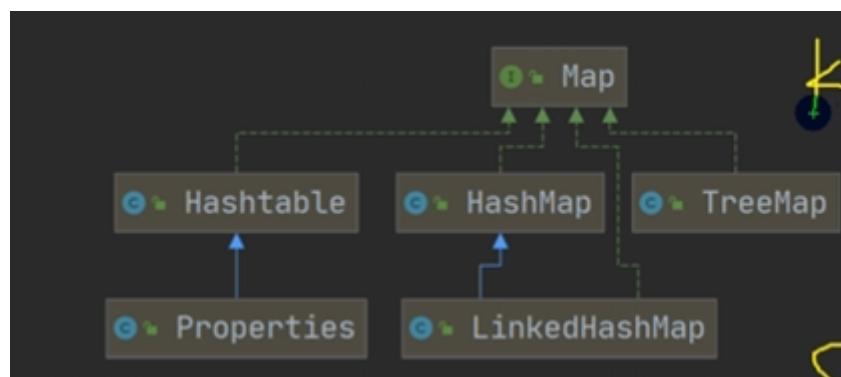
3 usages
class Person implements Comparable{
    @Override
    public int compareTo(Object o) {
        return 0;
    }
}

```

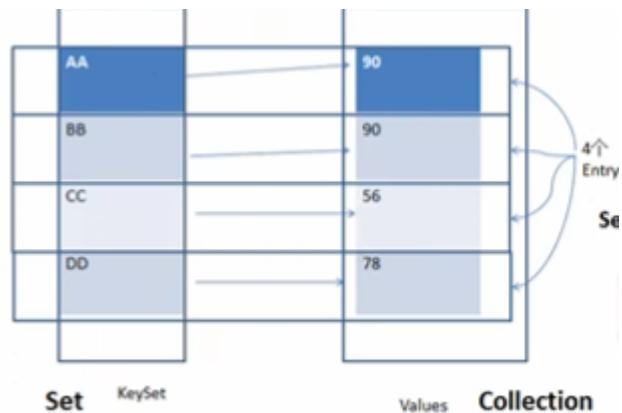
Map

实现的接口类有：Hashtable, HashMap和Properties,HashMap使用频率最高

1. key-value可以是任意类
2. key不允许重复，跟HashSet一样
3. 可以认为HashMap是完全版的HashSet， value值可变
4. key可以为null，但只能有一个； value则没有限制； key-value是一对一关系
5. 常用String类作为Map的key
6. 没有实现同步，**线程不安全**



源码



1. k-v 最后存储在 HashMap 的 newNode 中，Node (HashMap 的静态内部类) 实现了 Entry 接口

```
Node<K,V> newNode(int hash, K key, V value, Node<K,V> next) {
    return new Node<>(hash, key, value, next);
}

static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;

    Node(int hash, K key, V value, Node<K,V> next) {
        this.hash = hash;
        this.key = key;
        this.value = value;
        this.next = next;
    }

    ...
}

// 重要的entrySet用于遍历, 当然是拷贝传递
public Set<Map.Entry<K,V>> entrySet() {
    Set<Map.Entry<K,V>> es;
    return (es = entrySet) == null ? (entrySet = new EntrySet()) : es;
}

// 重要的keySet构造器, 同样是拷贝传递
public Set<K> keySet() {
    Set<K> ks = keySet;
    if (ks == null) {
        ks = new KeySet();
        keySet = ks;
    }
    return ks;
}

// 重要的values构造器, 同样是拷贝传递
public Collection<V> values() {
    Collection<V> vs = values;
    if (vs == null) {
        vs = new Values();
        values = vs;
    }
}
```

```
    return vs;
}
```

```
// Entry是k-v之母
interface Entry<K,V>{
    // 两个重要的方法，方便获得key和value
    K getKey();
    V getValue();
    // 两个重要的构造器获取对应的key数组和value数组
    Set<K> keySet();
    Collection<V> values();
    ...
}
```

2. k-v 为了方便程序员遍历，还会创建EntrySet集合（也是HashMap的内部类），集合元素类型是 Map.Entry，但实际上存的还是HashMap\$Node（实现了Entry接口，向上转型）；一个Entry对象就有k-v，这是拷贝传递，而不是值传递

```
map = {HashMap@503} "{hihi=hihi, 123=ss}"
  > ⚡ table = {HashMap$Node[16]@520}
  ↴ ⚡ entrySet = {HashMap$EntrySet@521} "[hihi=hihi, 123=ss]"
    > ⚡ this$0 = {HashMap@503} "{hihi=hihi, 123=ss}"
      > ⚡ table = {HashMap$Node[16]@520}
      > ⚡ entrySet = {HashMap$EntrySet@521} "[hihi=hihi, 123=ss]"
```

此处table与entrySet里面的table都是@520，说明指向的是同一个对象

```
// EntrySet的元素作为引用拷贝了指向Node的对象
final class EntrySet extends AbstractSet<Map.Entry<K,V>> {
    ...
}
```

3. 当把HashMap\$Node对象存放到entrySet时，因为Entry定义了getKey和getValue方法方便遍历
4. HashMap有EntrySet、KeySet、Values三个重要的内部类用于遍历

常用方法

都是按照key值来：增(add)删(remove)改(add)查(get或者containsKey)

MapMethod.java

- 1) put:添加
- 2) remove:根据键删除映射关系
- 3) get: 根据键获取值
- 4) size: 获取元素个数
- 5) isEmpty: 判断个数是否为0
- 6) clear: 清除
- 7) containsKey: 查找键是否存在

遍历

有entrySet、keySet、values

三者都继承了Collection，因而实现了Iterator

```
Map map = new HashMap();
map.put("jack", "tes");
map.put("uzi", "edg");
map.put("theshy", "wbg");
System.out.println();
// 第1类: keySet(继承了Collection, Collection实现了Iterable)
// 1.1: 增强for
Set keyset = map.keySet();
for (Object key: keyset){
    System.out.println("key = " + key + " value = " + map.get(key));
}
// 1.2: 使用iterator
Iterator iterator = keyset.iterator();
while (iterator.hasNext()){
    Object key = iterator.next();
    System.out.println("key = " + key + " value = " + map.get(key));
}
// 第2类: values (作为Collection实现了Iterable): 只能访问value
// 2.1: 增强for
Collection values = map.values();
for (Object value: values){
    System.out.println("value = " + value);
}
// 2.2: 使用iterator
Iterator iterator1 = values.iterator();
while (iterator1.hasNext()){
    Object value = iterator1.next();
    System.out.println("value = " + value);
}
// 第3类: entrySet(继承了Collection)
// 3.1 增强for
Set entryset = map.entrySet();
for (Object entry: entryset){
    // 需要向下转型
    // 因为需要访问私有方法
    Map.Entry entry1 = (Map.Entry)entry;
    System.out.println("key = " + entry1.getKey() + " value = " +
entry1.getValue());
}
// 3.2 迭代器
Iterator iterator2 = entryset.iterator();
while (iterator2.hasNext()) {
    Object next = iterator2.next();
    Map.Entry entry1 = (Map.Entry)next;
    System.out.println("key = " + entry1.getKey() + " value = " +
entry1.getValue());
}
```

HashMap

源码同HashSet

Hashtable

基本与HashMap一致，底层是Entry数组（实现了Map.Entry）

```
private static class Entry<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Entry<K,V> next;

    protected Entry(int hash, K key, V value, Entry<K,V> next) {
        this.hash = hash;
        this.key = key;
        this.value = value;
        this.next = next;
    }

    @SuppressWarnings("unchecked")
    protected Object clone() {
        return new Entry<>(hash, key, value,
                           (next==null ? null : (Entry<K,V>) next.clone()));
    }
}
```

1. k-v值均不能为空，会抛出NullPointerException
2. 线程安全(synchronized)
3. 效率比HashMap低

源码

rehash

扩容机制：初始大小为11

```
protected void rehash() {
    int oldCapacity = table.length;
    Entry<?,?>[] oldMap = table;

    // overflow-conscious code
    // 新容量 = 2*旧容量 + 1
    int newCapacity = (oldCapacity << 1) + 1;
    if (newCapacity - MAX_ARRAY_SIZE > 0) {
        if (oldCapacity == MAX_ARRAY_SIZE)
            // Keep running with MAX_ARRAY_SIZE buckets
            return;
        newCapacity = MAX_ARRAY_SIZE;
    }
    Entry<?,?>[] newMap = new Entry<?,?>[newCapacity];

    modCount++;
    threshold = (int) Math.min(newCapacity * loadFactor, MAX_ARRAY_SIZE + 1);
```

```

table = newMap;

for (int i = oldCapacity ; i-- > 0 ;) {
    for (Entry<K,V> old = (Entry<K,V>)oldMap[i] ; old != null ; ) {
        Entry<K,V> e = old;
        old = old.next;

        int index = (e.hash & 0x7FFFFFFF) % newCapacity;
        e.next = (Entry<K,V>)newMap[index];
        newMap[index] = e;
    }
}
}

```

Properties

继承了Hashtable，因此k-v都不能有null

TreeMap

比较

1. 存储类型：一组对象（单列）或一组键值对（双列）

Collection接口

1. 允许重复：List

1. 增删多：LinkedList(双向链表)
2. 改查多：ArrayList(维护Object类型的可变数组)

2. 不允许重复：Set

1. 无序：HashSet(底层是HashMap，维护了一个哈希表，数组+链表+红黑树)
2. 排序：TreeSet
3. 插入和取出顺序一致：LinkedHashSet(底层是LinkedHashMap，维护数组+双向链表)

Map

1. key无序：HashMap(底层是哈希表,jdk7:数组+链表，jdk8：数组+链表+红黑树)
2. key排序：TreeMap
3. key插入和取出顺序一致：LinkedHashMap
4. 读取文件：Properties

CH13 泛型(Generic)

在编译期间（实例化）就确定泛型E是什么类型

泛型是一种可以指定数据类型的数据类型

细节

1. 泛型只能是**引用类型** (如Integer) , 不能是基本数据类型(如int)
2. 指定泛型具体类型后, 可以传入其**子类**
3. 如果不指定泛型, **默认是Object**
4. 可以简写(List list = new List<>();)

自定义泛型类

1. 不能用泛型实例化数组
2. 一般用大写字母自定义泛型,E , K , V
3. 普通成员可以使用泛型, **静态成员**不能使用泛型, 因为静态与类相关, 类加载时已完成, **未实例化**
此时泛型还没有确定

自定义泛型接口

```
interface 接口名<T, R, ...>{  
}
```

1. 接口中**静态成员**不能使用泛型
2. 泛型接口的类型, 在**继承或者实现接口**时确定
3. 没有指定类型, 默认为Object

自定义泛型方法

```
修饰符 <T, R, ...>返回类型 方法名 (参数列表) {  
}
```

1. 泛型方法被调用时, 泛型具体类型已经确定
2. 修饰符后没有泛型说明<T, R>的方法不是泛型方法, 而是**方法使用了泛型**

```
// 方法使用了类的泛型  
public void run(E e) {  
    System.out.println(e.toString());  
}  
// 泛型方法  
public <E> void run(E e) {  
    System.out.println(e.toString());  
}  
// 使用了类的泛型的泛型方法  
public <E> void run(T t) {  
    System.out.println(e.toString());  
}
```

注意事项

1. 泛型没有继承性

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

2. 通配符

- <?>: 支持任意泛型
- <? extends A>: 可以接收A或者A的子类
- <? super A>: 可以接收A或者A的父类

```
public class GenericExtends {
    public static void main(String[] args) {
        Object o = new String("xx");
        List<Object> list = new ArrayList<>();
        List<String> list2 = new ArrayList<>();
        List<AA> list3 = new ArrayList<>();
        List<BB> list4 = new ArrayList<>();
        List<CC> list5 = new ArrayList<>();

        hihi(list);
        hihi(list2);
        hihi(list3);
        hihi(list4);
        hihi(list5);

        // 非A或者A的子类, 错误
        hihi2(list);
        // 非A或者A的子类, 错误
        hihi2(list2);
        hihi2(list3);
        hihi2(list4);
        hihi2(list5);

        hihi3(list);
        // 非A或者A的父类, 错误
        hihi3(list2);
        hihi3(list3);
        // 非A或者A的父类, 错误
        hihi3(list4);
        // 非A或者A的父类, 错误
        hihi3(list5);
    }

    public static void hihi(List<?> c) {
        for (Object o : c) {
            System.out.println(o);
        }
    }

    public static void hihi2(List<? extends AA> c) {
        for (Object o : c) {
            System.out.println(o);
        }
    }

    public static void hihi3(List<? super AA> c) {
        for (Object o : c) {
```

```

        System.out.println(o);
    }
}
}

class AA {
}

class BB extends AA {
}

class CC extends BB {
}

```

3.

```

HashMap<String, Student> hashMap = new HashMap();
Student stu1 = new Student("stu1", 12);
Student stu2 = new Student("stu2", 13);
hashMap.put(stu1.getName(), stu1);
hashMap.put(stu2.getName(), stu2);
Iterator<Map.Entry<String, Student>> iterator = hashMap.entrySet().iterator();
while (iterator.hasNext()) {
    Map.Entry<String, Student> next = iterator.next();
    System.out.println("name = " + next.getKey() + " value = " +
next.getValue());
}
for (String s : hashMap.keySet()) {
    System.out.println("name = " + s + " value = " + hashMap.get(s));
}

```

```

public class GenericExercise02 {
    public static void main(String[] args) {
        ArrayList<Employee> arrayList = new ArrayList<>();
        Employee hihi = new Employee("A", 30, new Mydate(2001, 13, 22));
        Employee hihi2 = new Employee("A", 30, new Mydate(2001, 15, 24));
        Employee hihi3 = new Employee("A", 30, new Mydate(2001, 10, 22));
        arrayList.add(hihi);
        arrayList.add(hihi2);
        arrayList.add(hihi3);
        arrayList.sort(new Comparator<Employee>() {
            @Override
            public int compare(Employee o1, Employee o2) {
                return o1.compareTo(o2);
            }
        });
        System.out.println(arrayList);
    }
}

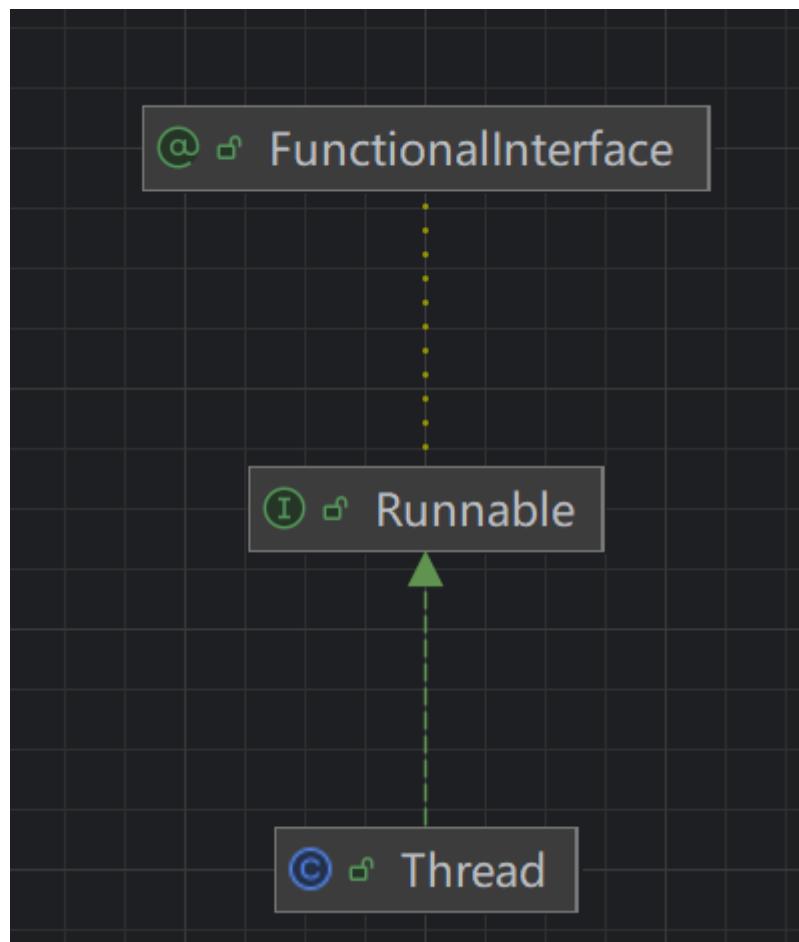
```

```
}

// 实现Comparable, 重写compareTo函数
class Employee implements Comparable<Employee>{
    @Override
    public int compareTo(Employee o2) {
        if (getName().compareTo(o2.getName()) != 0){
            return getName().compareTo(o2.getName());
        } else {
            return getBirthday().compareTo(o2.getBirthday());
        }
    }
    ...
}

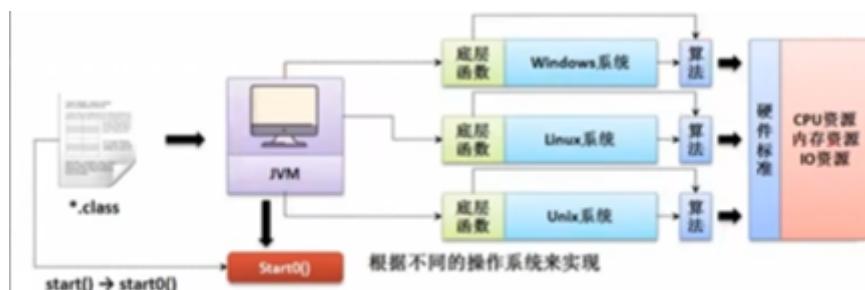
class Mydate implements Comparable<Mydate> {
    @Override
    public int compareTo(Mydate o2) {
        if (getYear() - o2.getYear() != 0) {
            return getYear() - o2.getYear();
        } else {
            if (getMonth() - o2.getMonth() != 0) {
                return getMonth() - o2.getMonth();
            } else {
                return getDay() - o2.getDay();
            }
        }
    }
    ...
}
```

CH14 线程



由进程创建的，是进程的实体，一个进程可以有多个线程

start会开启新栈，真正的多线程，run并不会，只会在原栈



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。

```
private native void start0();
```

start0方法是native，真正实现了多线

程，即底层jvm实现的

```
public synchronized void start() {
    if (threadStatus != 0)
        throw new IllegalThreadStateException();
    group.add(this);

    boolean started = false;
    try {
        // start0, 多线程
    }
```

```
        start0();
        started = true;
    } finally {
        try {
            if (!started) {
                group.threadStartFailed(this);
            }
        } catch (Throwable ignore) {
        }
    }
}
```

实现

继承Thread类

实现Runnable接口

静态代理模式

由于不能直接继承Thread，因而创建一个**Runnable的实现类**，再交由Thread构造器帮忙初始化

```
Dog dog = new Dog();
Thread thread = new Thread(dog);
thread.start();
// Thread构造器源码
public class Thread implements Runnable {
    public Thread(Runnable target) {
        init(null, target, "Thread-" + nextThreadNum(), 0);
    }
    ...
}
// 模拟的Thread类
class ThreadProxy implements Runnable{

    private Runnable target = null;
    @Override
    public void run() {
        if (target != null){
            target.run();
        }
    }

    public ThreadProxy(Runnable target) {
        this.target = target;
    }
    public void start(){
        start0();
    }
    public void start0(){
        run();
    }
}
```

```
// 实现了Runnable类  
class Dog implements Runnable{  
    ...  
}
```

区别

- 本质没区别，继承Thread也是通过实现了Runnable接口来实现构造器
- 实现Runnable更适合多线程共享一个资源的情况，建议使用Runnable

概念

并发

同一时刻多任务交替执行，造成“同时运行”的错觉：空分复用；单核CPU实现的任务就是并发

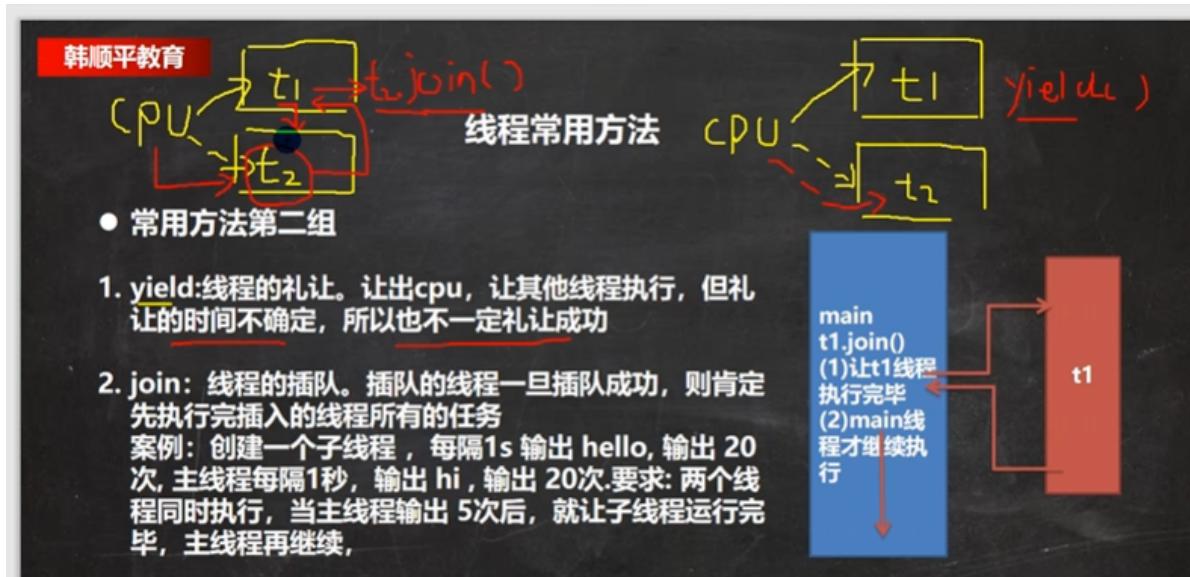
并行

同一时刻多个任务同时执行，时分复用

常用方法

1. `setName` //设置线程名称，使之与参数 name 相同
2. `getName` //返回该线程的名称
3. `start` //使该线程开始执行；Java 虚拟机底层调用该线程的 `start0` 方法
4. `run` //调用线程对象 `run` 方法；
5. `setPriority` //更改线程的优先级
6. `getPriority` //获取线程的优先级
7. `sleep` //在指定的毫秒数内让当前正在执行的线程休眠（暂停执行）
8. `interrupt` //中断线程

- `yield`: 不一定成功，看CPU资源调度
- `join`(并行 变串行)



守护线程

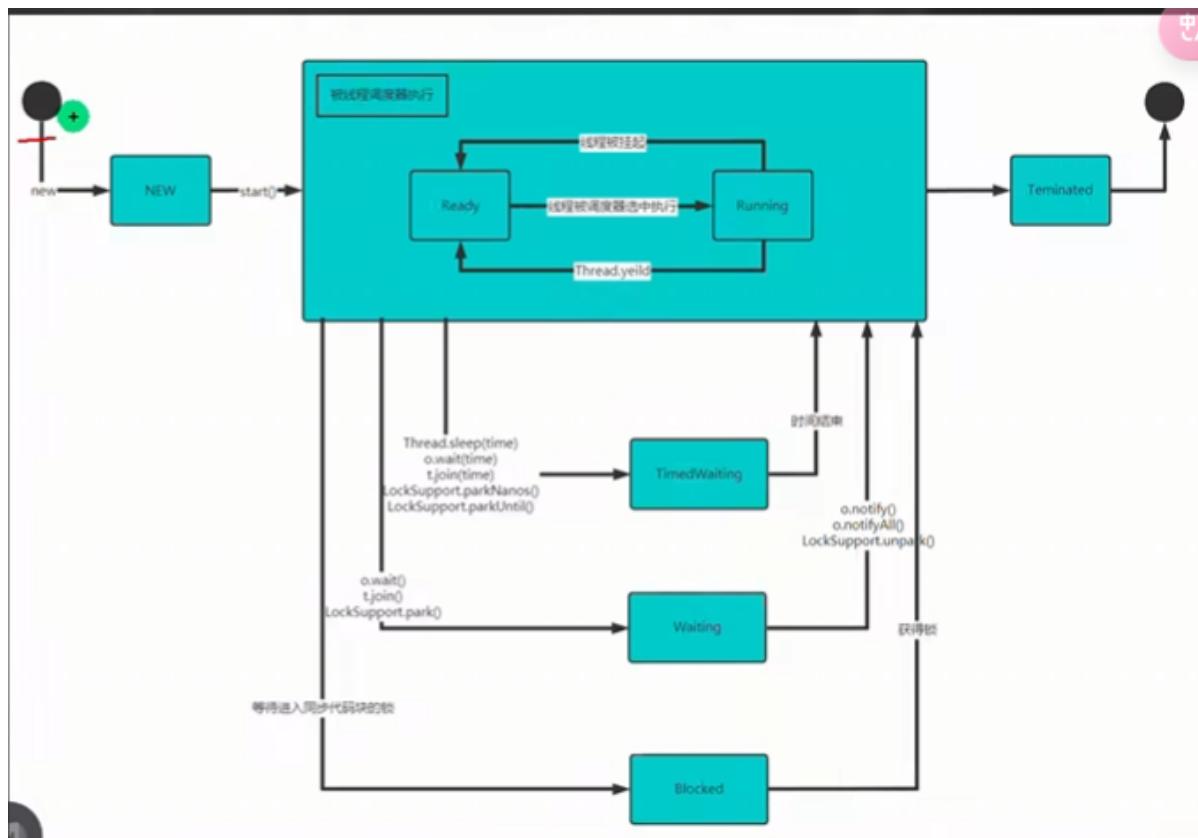
```
thread.setDaemon(true);
```

- 用户线程又叫工作线程，当线程的任务执行完或通知方式结束
- 守护线程一般为工作线程服务，当所有用户线程结束时，守护线程自动结束
- 常见的守护线程：垃圾回收机制

线程退出

```
class T extends Thread{  
    boolean loop = true;  
    @Override  
    public void run() {  
        while (loop){  
            setLoop(false);  
            System.out.println("hihi");  
            try {  
                Thread.sleep(50);  
            } catch (InterruptedException e) {  
                throw new RuntimeException(e);  
            }  
        }  
    }  
  
    public void setLoop(boolean loop) {  
        this.loop = loop;  
    }  
}
```

生命周期



- NEW(start之前)
- RUNNABLE(包含READY & RUNNING)
- WAITING(join)
- TIME_WAITING(sleep)
- BLOCKED
- TERMINATED

有7种状态， runnable有2种状态 ((ready准备)以及running(真的在run了))

synchronized

- 多线程中，一些敏感数据不允许被多个线程同时访问，此时就用同步访问技术，保证数据在**任何同一时刻，最多有一个线程访问（无法时分复用）**，以保证数据的完整性
- 当有一个线程对内存进行操作时，**其他线程都不可以对这个内存地址进行操作**，直到该线程完成操作，其他线程才可以对该内存地址进行操作

同步代码块

```
synchronized(对象){  
}  
static synchronized(类名.class){  
}
```

同步方法

访问修饰符 + synchronized + 返回类型 + 方法名

- 若没有static修饰，**默认锁对象为this**
- 若有static修饰，**默认锁对象为类名.class**

互斥锁

每个对象都对应于一个可称为“互斥锁”的标记，保证任意时刻只能有一个线程访问该对象，其实就是并行->串行

- 局限性：程序的执行效率降低
- 同步方法（非静态）的锁可以是**this**（当前对象），也是**其他对象**
- 如果是**静态**同步方法，则锁位**当前类本身**

```
class SellTicket03 implements Runnable {  
    private int ticketNum = 100;  
    private boolean loop = true;  
    Object object = new Object();  
    @Override  
    public void run() {  
        while(loop){  
            f();  
            System.out.println("窗口：" + Thread.currentThread().getName() + "售出  
一张票" +  
                "剩余票数：" + --ticketNum);  
        }  
    }  
    public synchronized void f() {  
        synchronized (object) {  
            if (ticketNum <= 0) {  
                loop = false;  
                System.out.println("售票结束");  
                return;  
            }  
        }  
        try {  
            Thread.sleep(50);  
        } catch (InterruptedException e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

线程死锁

o1,o2都为静态变量

- A线程在获得o1对象锁之后判断拿不拿得到o2锁，此时B线程已拿到o2的对象锁再判断拿不拿得到o1，卡在那里了

- 避免：不要嵌套写互斥锁

```

//下面业务逻辑的分析
//1. 如果flag 为 T, 线程A 就会先得到/持有 o1 对象锁, 然后尝试去获取 o2 对象锁
//2. 如果线程A 得不到 o2 对象锁, 就会Blocked
//3. 如果flag 为 F, 线程B 就会先得到/持有 o2 对象锁, 然后尝试去获取 o1 对象锁
//4. 如果线程B 得不到 o1 对象锁, 就会Blocked

if (flag) {
    synchronized (o1) { //对对象互斥锁, 下面就是同步代码
        System.out.println(Thread.currentThread().getName() + "进入1"); A
        synchronized (o2) { // 这里获得li对象的监视权
            System.out.println(Thread.currentThread().getName() + "进入2");
        }
    }
} else {
    synchronized (o2) {
        System.out.println(Thread.currentThread().getName() + "进入3"); B
        synchronized (o1) { // 这里获得li对象的监视权
            System.out.println(Thread.currentThread().getName() + "进入4");
        }
    }
}

```

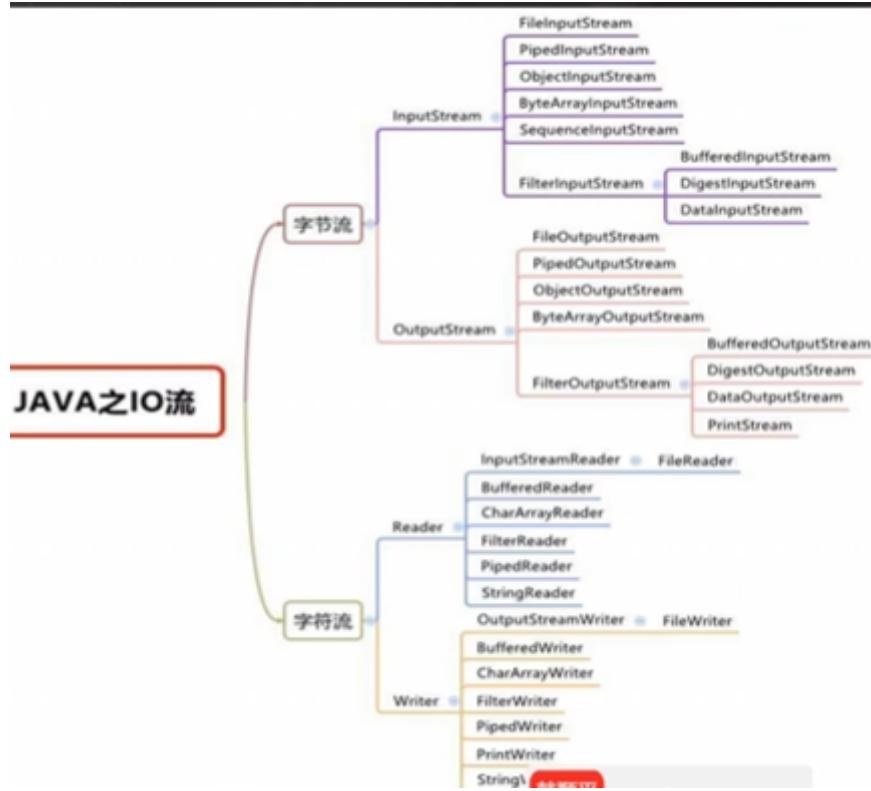
释放锁

- 同步代码块或同步方法执行结束, 自动释放
- 在**同步代码块、同步方法**遇到**break、return**语句
- 同步代码块、同步方法中出现了**未处理的Error或Exception**, 导致异常
- 同步代码块、同步方法中执行了线程对象的**wait()**方法, 线程暂停

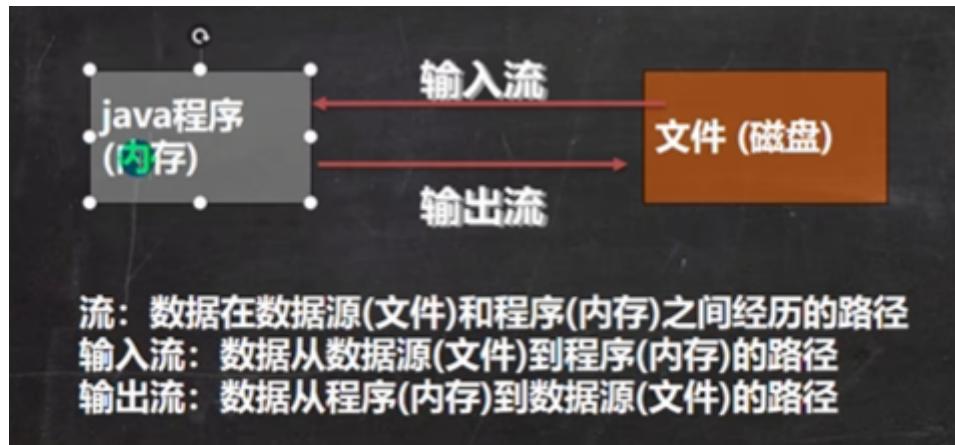
不释放锁的情况

- sleep、yield**, 暂停线程的执行, 不会释放锁
- 线程执行同步代码块时, **其他线程调用了该线程的suspend()**将该方法挂起 (处于Runnable中的Ready状态), 也不释放锁 (避免使用resume和suspend)

CH15 IO流



文件在程序中是以流的形式来操作的



输入与输出是以Java为中心的

常用方法 (File类)

构造器有多种创建方式：

1. 路径
2. 父路径 + 子路径
3. 父File + 子路径

● 创建文件对象相关构造器和方法

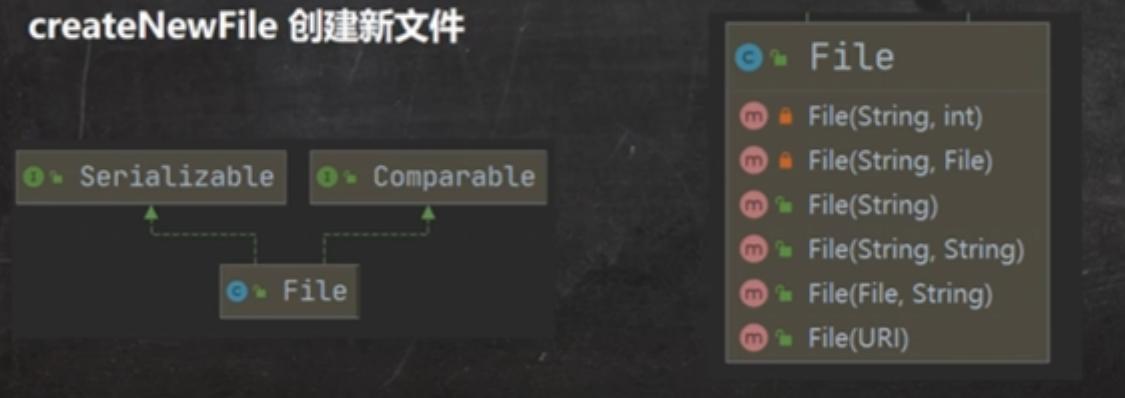
➤ 相关方法

`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

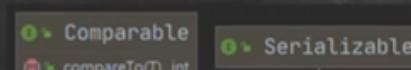
createNewFile 创建新文件



常用的文件操作

● 获取文件的相关信息

`getName`、`getAbsolutePath`、`getParent`、`length`、`exists`、`isFile`、`isDirectory`



● 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

<code>m File mkdir()</code>	<code>boolean</code>
<code>m File mkdirs()</code>	<code>boolean</code>

```
String filePath = "W:\\\\mytemp";
File file = new File(filePath);
if (file.mkdir()) {
    System.out.println("创建成功");
} else {
    System.out.println("已存在，创建失败");
}
File file1 = new File(file, "hello.txt");
// 如果文件不存在则创建文件
if (!file1.exists()) {
    // createNewFile才是创建文件，mkdir是创建文件夹
    if (file1.createNewFile()) {
        BufferedWriter bufferedWriter = null;
        try {
            bufferedWriter = new BufferedWriter(new FileWriter(file1));
            bufferedWriter.write("helloworld好好好");
        } catch (IOException e) {

```

```

        throw new RuntimeException(e);
    } finally {
        try {
            bufferedWriter.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
    System.out.println(file1 + "创建成功");
}
} else {
    System.out.println(file1 + "已经存在，创建失败");
}

```

分类

The diagram illustrates the classification of Java I/O streams. It features a vertical red line on the right labeled '节点流' (Node Stream) at the top and '处理流' (Processing Stream) at the bottom. A horizontal red box highlights the first two columns of the table.

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

操作数据单位

- 字节流 (8 bit) : 应用二进制
- 字符流 (按字符)

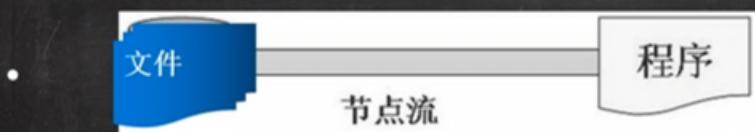
按数据流的流向

均为抽象类，要使用要通过继承其子类

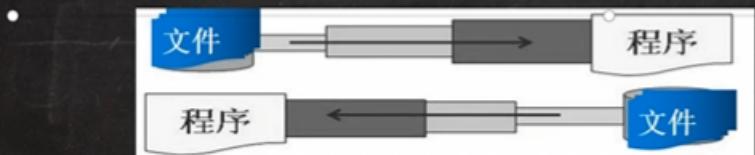
- 输入流(InputStream, Reader)
- 输出流(OutputStream, Writer)

按流的角色不同

1. 节点流可以从一个特定的数据源读写数据，如FileReader、FileWriter [源码]



2. 处理流(也叫包装流)是“连接”在已存在的流（节点流或处理流）之上，为程序提供更为强大的读写功能，如BufferedReader、BufferedWriter [源码]

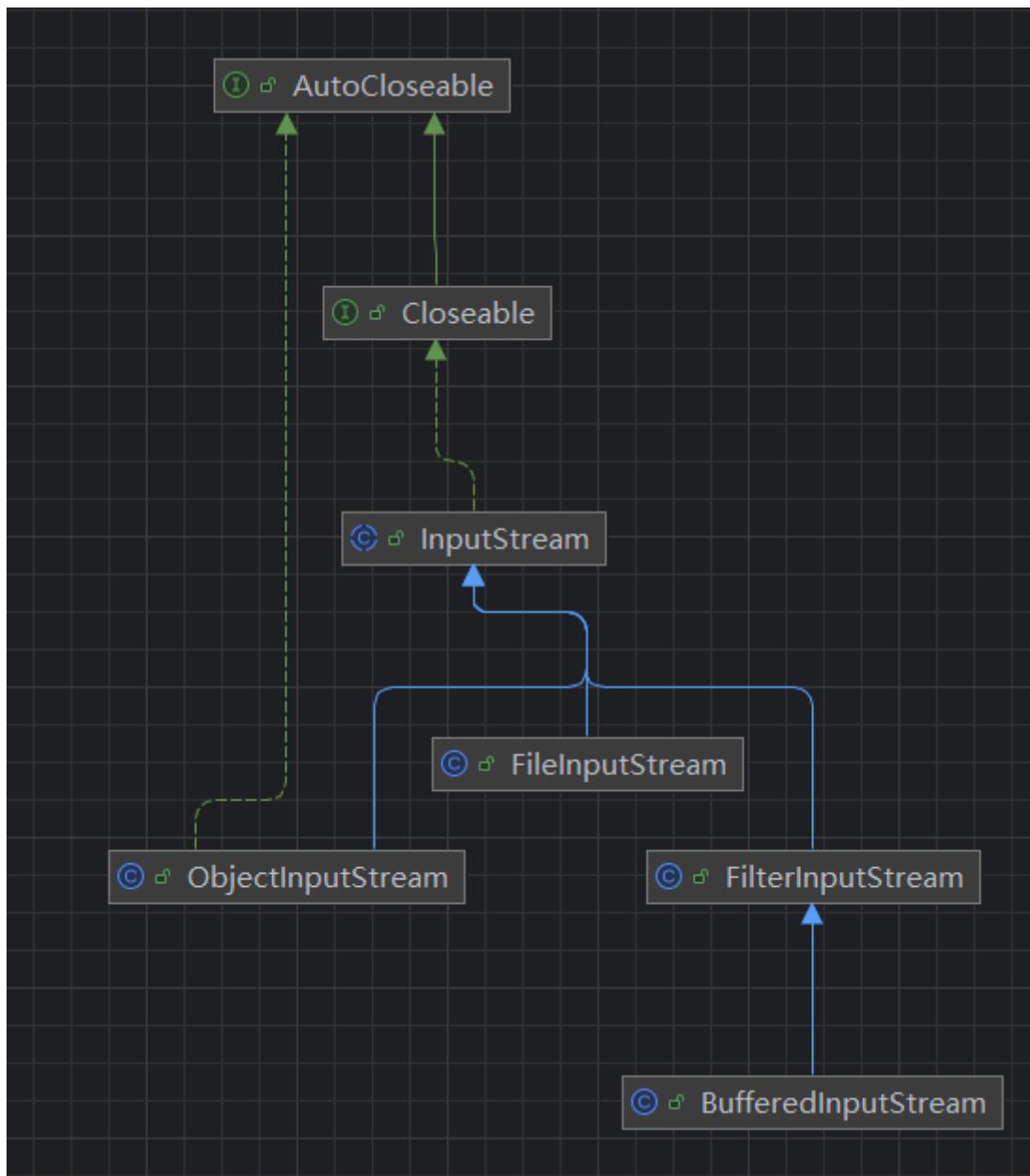


- **节点流**: 比较局限，对指定文件类型进行处理的(FileReader, ByteArrayReader, PipedReader; FileInputStream, ByteArrayInputStream, PipedInputStream)
- 处理流/包装流(BufferedReader/BufferedWriter): 例如BufferedReader内含Reader属性，可以封装任意一个**节点流**，节点流可以是任意的，**只要是Reader子类**
连接在**已存在的流（节点流或处理流）**之上，为程序提供更强大的读写功能

```
① public class BufferedReader extends Reader {  
  
    private Reader in;  
  
    private char cb[];
```

```
② BufferedReader  
③ BufferedReader(Reader, int)  
④ BufferedReader(Reader)
```

字节流(InputStream和OutputStream)



FileInputStream

文件输入流

- `read(byte[])`: 读取指定的字节数, 返回的是实际读取的字节数
- `read()`: 单字节读取
- `close()`: 用完记得关闭流, 防止堵塞

```
public static void readFile01(){
    String path = "w:\\temp\\1.txt";
    int readData = 0;
    byte[] buf = new byte[8];
    int readLen = 0;
    FileInputStream fileInputStream = null;
    try {
        fileInputStream = new FileInputStream(path);
        while ((readLen = fileInputStream.read(buf)) != -1) {
            System.out.println(new String(buf, 0, readLen));
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (fileInputStream != null) {
            fileInputStream.close();
        }
    }
}
```

```

        }
    } catch (IOException e) {
        throw new RuntimeException(e);
    } finally {
        // 关闭文件流，释放资源
        try {
            fileInputStream.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}

```

FileOutputStream

- write(char[])
- write(char[], int offset, int len): 推荐这个，指定长度更安全

```

// 若append置为true，则在文件末端写入而不是直接覆盖
public FileOutputStream(String name, boolean append)
    throws FileNotFoundException
{
    this(name != null ? new File(name) : null, append);
}

```

```

String filePath = "w:\\\\temp\\\\1.txt";
String str = "hello world";
FileOutputStream fileOutputStream = null;
try {
    fileOutputStream = new FileOutputStream(filePath);
    // String转为字节数组
    fileOutputStream.write(str.getBytes());
} catch (IOException e) {
    throw new RuntimeException(e);
} finally {
    try {
        fileOutputStream.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

```

文件复制

```

String filePath = "w:\\\\temp\\\\1.jpg";
String destPath = "w:\\\\temp\\\\2.jpg";
FileInputStream fileInputStream = null;
FileOutputStream fileOutputStream = null;

try {
    fileInputStream = new FileInputStream(filePath);
    fileOutputStream = new FileOutputStream(destPath);
    byte[] buf = new byte[1024];

```

```

int readLen = 0;
while ((readLen = fileInputStream.read(buf)) != -1){
    // 一定要用这个方法
    fileOutputStream.write(buf, 0, readLen);
}
} catch (IOException e) {
    throw new RuntimeException(e);
}finally {
    try {
        fileInputStream.close();
        fileoutputStream.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
}

```

字符流 (Reader和Writer)



常用方法

基本与字节流一致，不同的在于使用FileWriter之后必须使用close()或者flush()，说明是在缓冲区

● FileWriter常用方法

- 1) new FileWriter(File/String): 覆盖模式，相当于流的指针在首端
- 2) new FileWriter(File/String,true): 追加模式，相当于流的指针在尾端
- 3) write(int):写入单个字符
- 4) write(char[]):写入指定数组
- 5) write(char[],off,len):写入指定数组的指定部分
- 6) write (string) : 写入整个字符串
- 7) write(string,off,len):写入字符串的指定部分

相关API: String类: toCharArray:将String转换成char[]

➢ 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!

FileReader

```
public static void readFile01(){  
    String filePath = "W:\\temp\\1.txt";  
    FileReader fileReader = null;  
    int data = ' ';  
    // 单字符读取  
    try {  
        fileReader = new FileReader(filePath);  
        while ((data = fileReader.read()) != -1){  
            System.out.println((char) data);  
        }  
    } catch (IOException e) {  
        throw new RuntimeException(e);  
    }finally {  
        if (fileReader != null){  
            try {  
                fileReader.close();  
            } catch (IOException e) {  
                throw new RuntimeException(e);  
            }  
        }  
    }  
}  
  
public static void readFile02(){  
    String filePath = "W:\\temp\\1.txt";  
    FileReader fileReader = null;  
    int readLen = 0;  
    char[] buf = new char[8];  
    // 按指定字符数组读取  
    try {  
        fileReader = new FileReader(filePath);  
        while ((readLen = fileReader.read(buf)) != -1){  
            System.out.println(new String(buf, 0, readLen));  
        }  
    } catch (IOException e) {  
        throw new RuntimeException(e);  
    }finally {  
        if (fileReader != null){  
            try {  
                fileReader.close();  
            } catch (IOException e) {  
                throw new RuntimeException(e);  
            }  
        }  
    }  
}
```

```

        try {
            fileReader.close();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    }
}

```

FileWriter

可以把char[]等价理解成String

- write(): 单字符
- write(char[]): 指定字符数组
- write(char[], int offset, int len): 指定字符数组长度
- write(String): 指定字符串
- write(String, int offset, int len): 指定字符串长度

```

String path = "w:\\temp\\1.txt";
String path2 = "w:\\temp\\2.txt";
FileWriter filewriter = null;
FileReader fileReader = null;
char[] buf = new char[8];
int readLen = 0;
try {
    fileReader = new FileReader(path);
    filewriter = new FileWriter(path2);
    while ((readLen = fileReader.read(buf)) != -1){
        filewriter.write(buf, 0, readLen);
    }
} catch (IOException e) {
    throw new RuntimeException(e);
} finally {
    try {
        // 一定要close!
        fileReader.close();
        filewriter.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}

```

流处理设计模式

```

// 主程序
public class Test {
    public static void main(String[] args) {
        // 处理低级流1
        BufferedReader_ bufferedReader_ = new BufferedReader_(new FileReader_());
        bufferedReader_.readFiles();
        // 处理低级流2
    }
}

```

```
        BufferedReader_ bufferedReader_2 = new BufferedReader_(new
StringReader_());
        bufferedReader_.readString();
    }
}
// 抽象父类
public abstract class Reader_ {
    public void readFile(){

    };
    public void readString(){

    };
}
// 低级（底层）流1
public class FileReader_ extends Reader_{
    // 重写方法：自己的特色
    @Override
    public void readFile(){
        System.out.println("对文件进行处理");
    }
}
// 低级（底层）流2
public class StringReader_ extends Reader_{
    @Override
    public void readString(){
        System.out.println("读取字符串");
    }
}
// 处理（包装）流
public class BufferedReader_ extends Reader_ {
    // 属性中含低级流
    private Reader_ reader_;

    public BufferedReader_(Reader_ reader_){
        this.reader_ = reader_;
    }
    // 动态绑定
    public void readFiles(int num){
        for (int i = 0; i < 10; i++) {
            reader_.readFile();
        }
    }
    // 动态绑定
    public void readStrings(int num){
        for (int i = 0; i < 10; i++) {
            reader_.readString();
        }
    }
}
```

处理流

注意：BufferedReader和BufferedWriter是按字符处理的，因此不要用于读取二进制文件（声音、视频、pdf、doc....）

BufferedReader

```
public static void main(String[] args) throws Exception {
    String filePath = "W:\\Code\\javacode\\chapter03\\Homework04.java";
    BufferedReader bufferedReader = new BufferedReader(new FileReader(filePath));
    String line;
    while((line = bufferedReader.readLine()) != null){
        System.out.println(line);

    };
    // 只需要关闭bufferedReader底层会自动关闭节点流
    bufferedReader.close();
}
```

BufferedWriter

```
//1.创建BufferedWriter 对象
BufferedWriter bw = new BufferedWriter(new FileWriter("src\\copy1.txt"));
//2.写入
bw.write("hello,韩顺平教育!");
//bw.write("\r\n");
bw.newLine();//插入一个和系统相关的换行符
bw.write("hello,java工程师~");
//3.关闭
bw.close();
```

```
public static void main(String[] args) throws Exception {
    String filePath = "W:\\Code\\javacode\\chapter03\\FloatDetail.java";
    String filePath2 = "W:\\Code\\javacode\\chapter03\\copy.java";
    BufferedReader bufferedReader = new BufferedReader(new FileReader(filePath));
    BufferedWriter bufferedWriter = new BufferedWriter(new
FileWriter(filePath2));
    String line;
    while((line = bufferedReader.readLine()) != null){
        // 完成copy操作
        bufferedWriter.write(line);
        // 根据系统插入换行符
        bufferedWriter.newLine();
    };
    bufferedReader.close();
    bufferedWriter.close();
}
```

BufferedInputStream

BufferedOutputStream

复制二进制文件，实际上读取字符也行，毕竟是底层的二进制

```
String srcPath = "w:\\temp\\1.mkv";
String destPath = "w:\\temp\\2.mkv";
BufferedInputStream bufferedInputStream = null;
BufferedOutputStream bufferedOutputStream = null;
byte[] buff = new byte[1024];
int readLen = 0;
try {
    bufferedInputStream = new BufferedInputStream(new FileInputStream(srcPath));
    bufferedOutputStream = new BufferedOutputStream(new
FileOutputStream(destPath));
    while ((readLen = bufferedInputStream.read(buff)) != -1){
        bufferedOutputStream.write(buff, 0, readLen);
    }
} catch (IOException e) {
    throw new RuntimeException(e);
} finally {
    try {
        bufferedInputStream.close();
        bufferedOutputStream.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

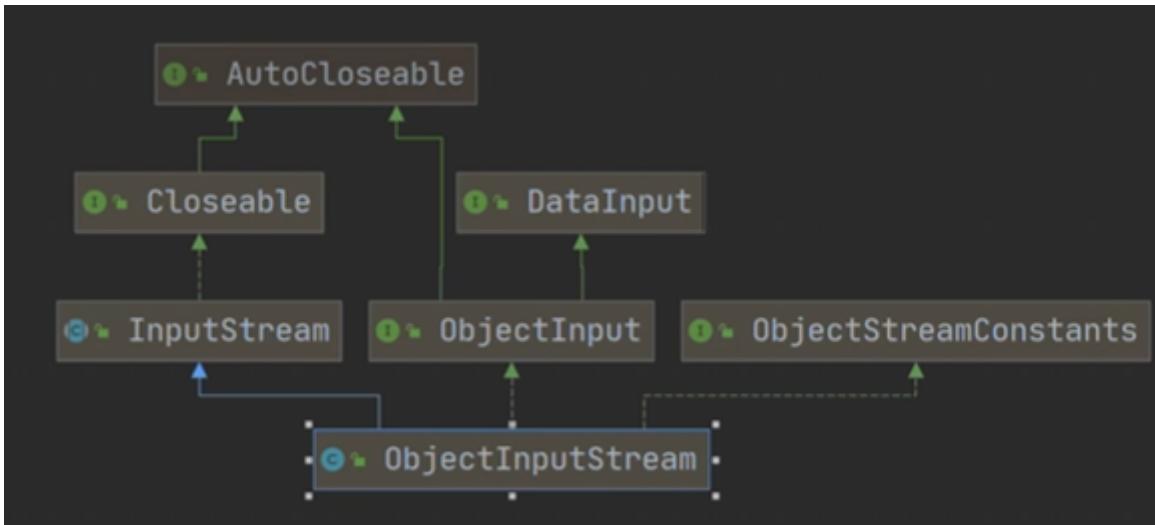
ObjectInputStream

提供序列化功能

```
public static void main(String[] args) throws IOException, ClassNotFoundException
{
    String filePath = "w:\\temp\\data.dat";
    // 读取序列化顺序一定要与output读取的时候一致！
    ObjectInputStream objectInputStream = new ObjectInputStream(new
FileInputStream(filePath));
    System.out.println(objectInputStream.readInt());
    System.out.println(objectInputStream.readBoolean());
    System.out.println(objectInputStream.readChar());
    System.out.println(objectInputStream.readUTF());
    // 要访问私有属性要通过向下转型
    Object o = objectInputStream.readObject();
    System.out.println("运行类型是" + o.getClass());
    // 如果希望引用dog
    // 则需要把dog放在同一个package中
    Dog dog = (Dog) o;
    System.out.println(dog.getName());
    objectInputStream.close();
}
```

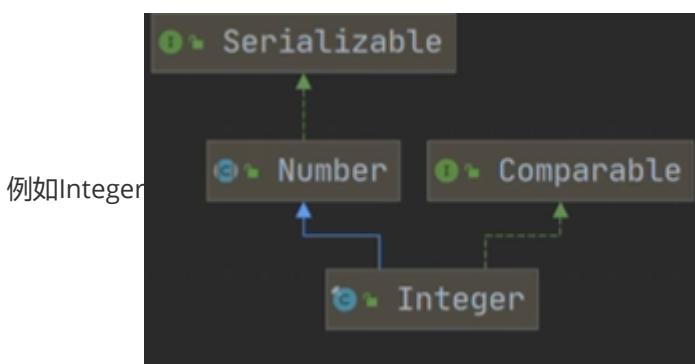
ObjectOutputStream

提供反序列化功能



序列化

1. 序列化：保存数据时，保存**数据类型和数据值**
2. 反序列化：恢复数据时，恢复**数据类型和数据值**
3. 要使某个类能序列化，要实现**Serializable**（推荐，标记接口，无任何成员）和**Externalizable**接口（一般不推荐，需要实现2个方法）
4. 序列化类中建议加入**serialVersionUID**，提高序列化兼容性
5. 序列化会默认序列化所有成员，**transient**和**static**除外
6. 序列化对象时，要求里面**属性的类型**也实现序列化接口
7. 序列化具有可继承性，本质是接口的继承



标准输入和输出流

- `System.in` 运行类型是**BufferedInputStream**，默认设备是键盘
- `System.out` 运行类型是**PrintStream**，默认设备是显示器

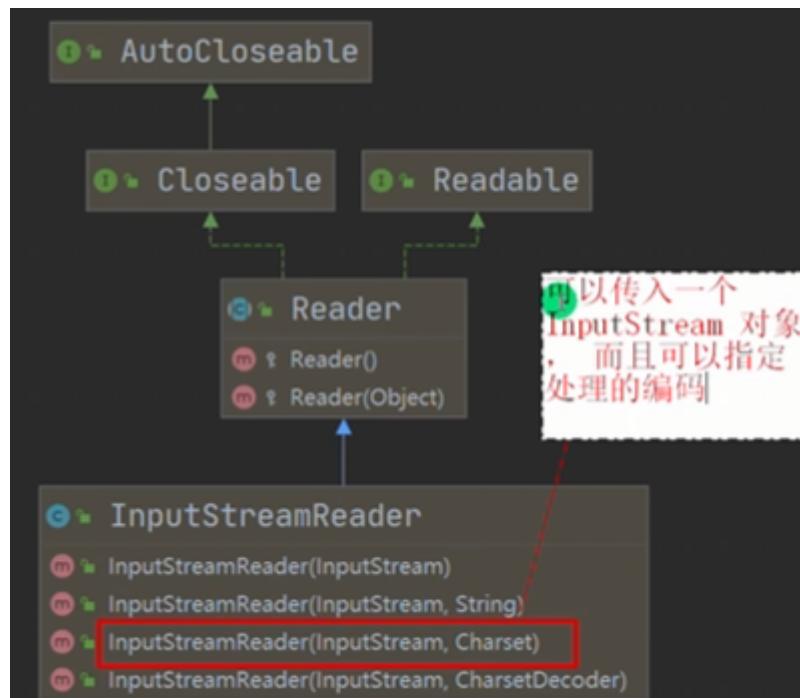
```
public final static InputStream in = null;
```

```
public final static PrintStream out = null;
```

转换流

InputStreamReader

将字节流包装成字符流，可以用于指定编码格式(utf-8, gbk...)



可以传入一个
InputStream 对象
，而且可以指定
处理的编码

```
String filePath = "w:\\temp\\1.txt";
BufferedReader br = new BufferedReader(new InputStreamReader(
    new FileInputStream(filePath), "utf-8"));
String s = br.readLine();
System.out.println(s);
br.close();
```

OutputStreamWriter

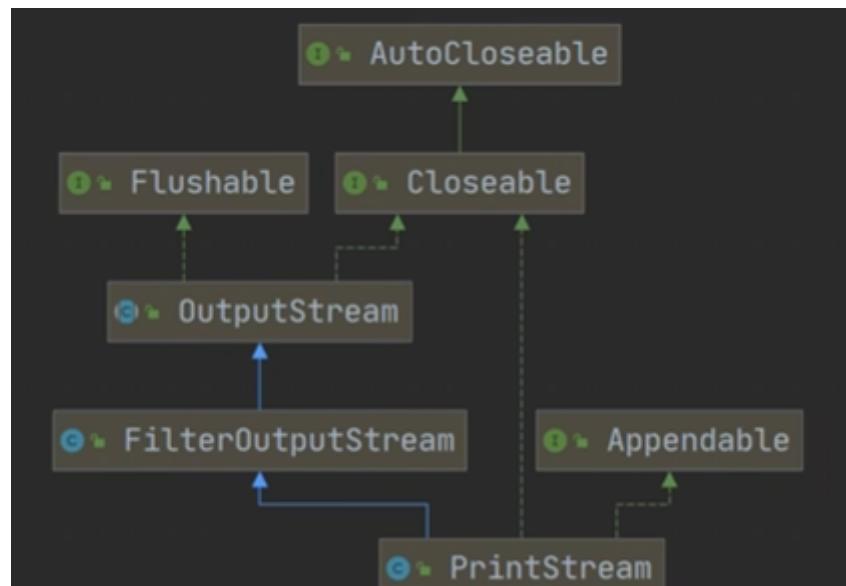
字节流转成（包装）字符流

```
String filePath = "w:\\temp\\a2.txt";
try {
    // 直接FileOutputStream
    OutputStreamWriter outputStreamWriter = new OutputStreamWriter(new
FileOutputStream(filePath), "gbk");
    outputStreamWriter.write("hihi,苏大");
    outputStreamWriter.close();
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

打印流

PrintStream

只有输出流没有输入流



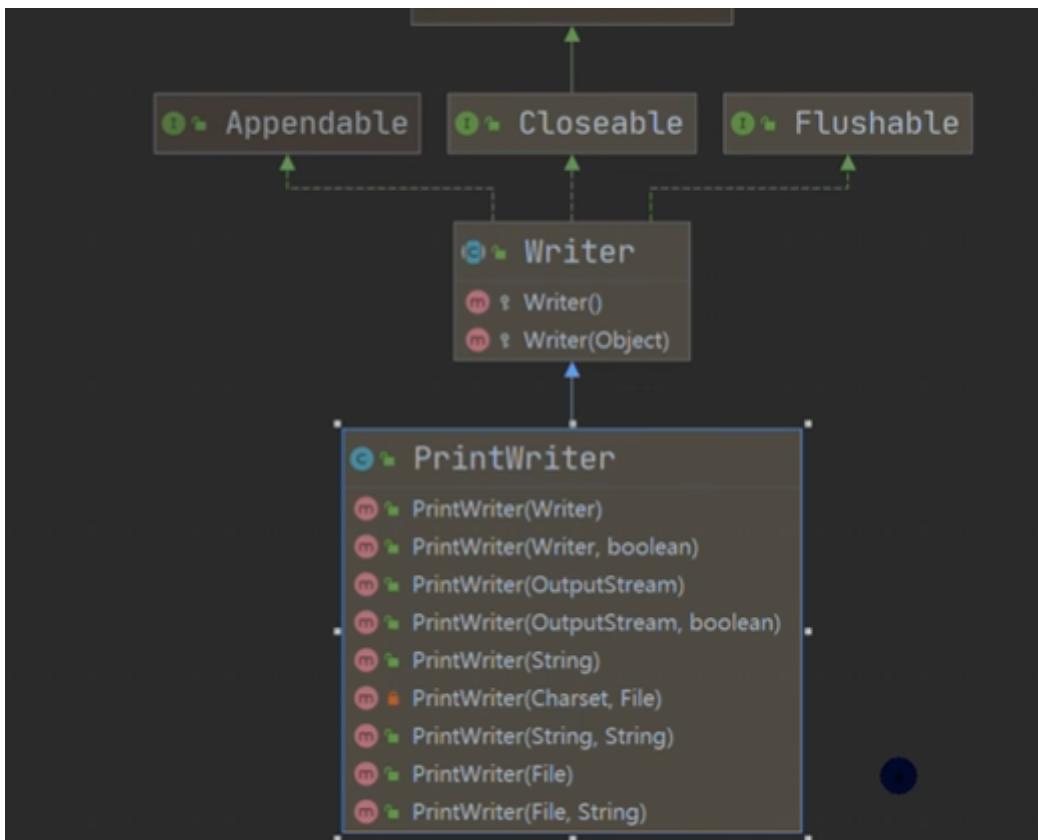
`System.out`本质就是`PrintStream`, 其中print里面源码用的是write(接受的是字节), 因此print和write效果一致

```
System.out.println(System.out.getClass());
PrintStream out = System.out;

out.print("hihi");
// 需先转换为字节
out.write("hihi2".getBytes());
out.close();
// setOut更改System.out的输出位置
System.setOut(new PrintStream("w:\\temp\\a2.txt"));
System.out.println("hello2222");
```

```
public void print(String s) {
    if (s == null) {
        s = "null";
    }
    write(s);
}
```

PrintWriter



```
PrintWriter printwriter = new PrintWriter(new FileWriter("w:\\\\a1.txt"));
printwriter.print("hihi上述");
// 一定要记得close!
printwriter.close();
```

Properties

继承自Hashtable

```
public static void main(String[] args) throws IOException {
    // 读取
    Properties properties = new Properties();
    // load读取Reader或InputStream对象
    properties.load(new FileReader("src\\\\mysql.properties"));
    // 显示k-v对
    properties.list(System.out);
    // 显示key值"pwd"对应的value值
    System.out.println(properties.get("pwd"));
    // 写入/修改
    Properties properties1 = new Properties();
    // 没有就是创建, 有就是修改, 因为本质是hashtable
    properties1.setProperty("charset", "utf-8");
    properties1.setProperty("user", "杰克");
    // null处是注释, 在properties的第一行
    properties1.store(new FileOutputStream("src\\\\mysql2.properties"), null);
}
```

```

public class Homework03 {
    public static void main(String[] args) throws IOException {
        Properties properties = new Properties();
        // 写入properties
        String filePath = "src\\mysql3.properties";
        properties.setProperty("name", "tom");
        properties.setProperty("age", "5");
        properties.setProperty("color", "red");
        properties.store(new FileOutputStream(filePath), "comment222");
        Properties properties1 = new Properties();
        // 读取properties属性用于创建Dog03对象
        properties1.load(new FileReader(filePath));
        Dog03 dog03 = new Dog03(properties1.getProperty("name"), new
        Integer(properties1.getProperty("age")), properties1.getProperty("color"));
        String filePath2 = "src\\dog.dat";
        // 使用ObjectOutputStream保存Dog03对象
        ObjectOutputStream objectOutputStream = new ObjectOutputStream(new
        FileOutputStream(filePath2));
        objectOutputStream.writeObject(dog03);
        objectOutputStream.close();
    }
}

class Dog03 implements Serializable{
    private String name;
    private int age;
    private String color;

    public Dog03(String name, int age, String color) {
        this.name = name;
        this.age = age;
        this.color = color;
    }
}

```

杂项

```

public class ObjectOutputStream_ {
    public static void main(String[] args) throws IOException {
        String filePath = "w:\\temp\\data.dat";
        ObjectOutputStream objectOutputStream = new ObjectOutputStream(new
        FileOutputStream(filePath));
        objectOutputStream.write(100);
        objectOutputStream.writeBoolean(true);
        objectOutputStream.writeChar('a');
        objectOutputStream.writeUTF("hihi");
        objectOutputStream.writeObject(new Dog("炫神", 9));
        // 记得关闭流
        objectOutputStream.close();
    }
}

```

```

class Dog implements Serializable {
    private String name;
    private int age;

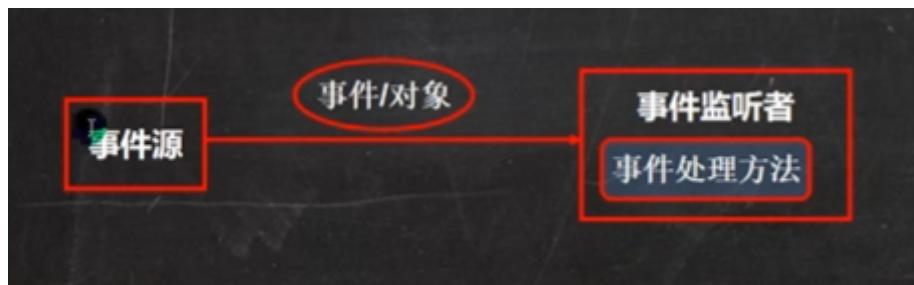
    public Dog(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return "Dog{" +
            "name='" + name + '\'' +
            ", age=" + age +
            '}';
    }
}

```

#

事件处理机制

事件源+事件（实现eventListener接口）

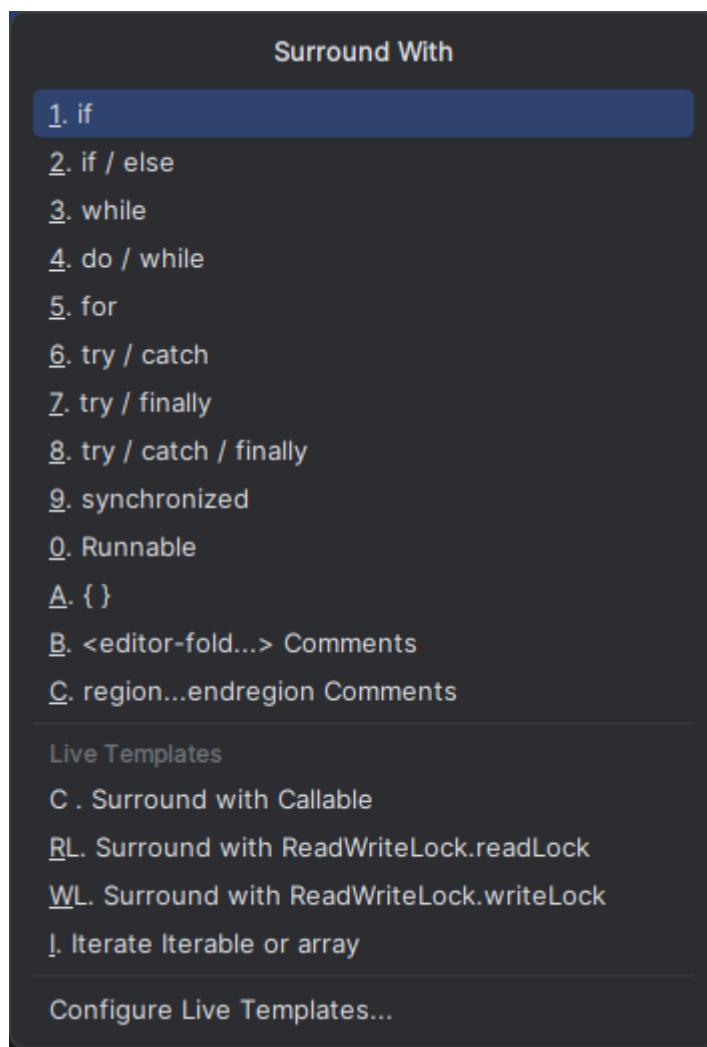


#

IDEA

Keymap

- / ** : 注释方法中的参数
- ctrl + D 复制当前行
- ctrl + alt + L 格式化
- **ctrl + shift + [生成**构造器或者get、 set封装方法
- ctrl + h 查看继承关系
- var或者alt enter 自动变量名
- ctrl + y 删除一行
- ctrl + alt + t(要先选中代码块): 插入语句



shift

- shift + enter 开启新一行
- ctrl + alt + enter 在上一行开启新一行

阅读源码

- alt + 7 : 查看类结构
- ctrl + h : 查看层次
- ctrl + n: 快速检索 (class)
- ctrl shift f: 全局文本检索
- ctrl + alt + b: 查看方法/类的实现类
- alt + f7: 查看方法被使用的情况
- ctrl + e: 查看最近使用的文件

模板(Settings -> editor -> Live templates)

ctrl + j 显示所有模板快捷方式

- main
- sout
- fori
- itit: 迭代器

Debug

- F7: 跳入 (某个方法)
- F8: 逐行执行
- shift + F8(跳出方法)
- F9: RESUME到下一个断点
- alt + shift + F7 : 强制跳入, 一般用于看源代码

动态传参

其他工具

JUnit

```
public class JUnit {  
    public static void main(String[] args) {  
  
    }  
    @Test  
    public void m1(){  
        System.out.println("m1");  
    }  
}
```