定义类

- 格式:

```
public class 类名{
    // 成员变量
    // 构造方法
    // 成员方法
    // 静态代码块
    // 构造代码块
    // 内部类
}
```

- 案例:

```
package com.itheima.demo1_定义类;

/**
 * @Author: pengzhilin
 * @Date: 2021/4/23 9:04
 */
public class Person {
    // 成员变量:非静态,静态
    private String name;
    private int age;

    // 构造方法
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Person() {
    }

    // 成员方法:非静态,静态
    public void show() {
        System.out.println("name:" + name + ",age:" + age);
    }

    // 静态代码块
    static {
        System.out.println("静态代码块随着类的加载而执行,并且只执行一次");
    }

    // 构造代码块
    {
        System.out.println("每次调用构造方法都会执行一次,并且在构造方法之前执行...");
    }

    // 内部类
    class NClass{
```

```
    }
}
```

对象的创建和使用

- 对象的创建:
  - 调用构造方法: `类名  对象名 = new 类名(实参);`
  - 调用静态方法: `类名 对象名 = 类名.静态方法名(实参);` `eg: Calendar,Executors...`
- 对象的使用:
  - 访问成员变量: `对象名.成员变量名`
  - 访问成员方法:
    - 无返回值的方法: `对象名.方法名(实参);`
    - 有返回值的方法:
      - 直接调用: `对象名.方法名(实参);`
      - **赋值调用:** `数据类型 变量名 = 对象名.方法名(实参);` **数据类型; 方法返回值类型**
      - **赋值调用:** `父类类型 变量名 = 对象名.方法名(实参);` **父类类型: 方法返回值类型的父类**
      - **赋值调用:** `子类类型 变量名 = (子类类型)对象名.方法名(实参);` **子类类型:方法返回值类型的子类**
      - 其他有参数方法的小括号中调用: `对象名.方法名(对象名.方法名(实参))`
        - eg:`System.out.println(对象名.方法名(实参));`
  - 访问静态方法: `类名.静态方法名(实参);`
    - 无返回值的方法: `类名(实参);`
    - 有返回值的方法:
      - 直接调用: `类名.方法名(实参);`
      - **赋值调用:** `数据类型 变量名 = 类名.方法名(实参);` **数据类型; 方法返回值类型**
      - **赋值调用:** `父类类型 变量名 = 类名.方法名(实参);` **父类类型: 方法返回值类型的父类**
      - **赋值调用:** `子类类型 变量名 = (子类类型)类名.方法名(实参);` **子类类型:方法返回值类型的子类**
      - 其他有参数方法的小括号中调用: `对象名.方法名(类名.方法名(实参))`
        - eg:`System.out.println(对象名.方法名(实参));`
- 案例:

```java
public class Test {
    public static void main(String[] args) {
        // 创建Person类的对象
        Person p1 = new Person();
        Person p2 = new Person("李四", 19);

        // 调用方法
        p1.show();
        p2.show();
    }
}
```

继承

- 格式:

```java
public class 子类名 extends 父类名{}
// 类的继承只能单继承,不能多继承,但可以多层继承
```

- 继承后的成员访问特点:
  - 子类会继承父类所有的成员变量和成员方法
  - 子类只能直接访问父类的非私有成员变量和成员方法
- 方法的重写:
  - 概述: 指的是父子类中出现一模一样的方法(返回值类型,方法名,参数列表,子类权限不能低于父类的权限)
  - 快捷键:
  - `alt+回车`----->重写抽象父类,接口的抽象方法
    - 直接写方法名,选中回车--->重写普通父类的方法

多态

多态的几种表现形式

- 多态的条件:
  - 继承 或者 实现
  - 父类的引用指向子类的对象 或者 接口的引用指向实现类的对象
  - 方法重写

- 多态的表现形式:
  - 普通父类多态

```
public class Animal{}
public class Dog extends Animal{}
public class Test{
    Animal anl = new Dog();
}
```

- 抽象父类多态

```
public abstract class Animal{}
public class Dog extends Animal{}
public class Test{
    Animal anl = new Dog();
}
```

- 父接口多态

```
public interface Animal{}
public class Dog implements Animal{}
public class Test{
    Animal anl = new Dog();
}
```

多态时访问成员的特点

- 多态是访问成员变量: 编译看左边,运行看左边
- 多态时访问成员方法:
  - 静态: 编译看左边,运行看左边
  - 非静态: 编译看左边,运行看右边

多态的应用场景:

- 变量多态

```
public class Animal{}
public class Dog extends Animal{}
public class Test{
    public static void main(String[] args){
        Animal anl = new Dog();
    }
}
```

- 形参多态------>重点

```
public class Animal{}
public class Dog extends Animal{}
public class Test{
    public static void main(String[] args){
        method(new Dog());
    }

    // 参数为父类类型,就可以接收任意子类对象,或者该父类类型的对象
    public static void method(Animal anl){

    }
}
```

- 返回值多态------>重点

```
public class Animal{}
public class Dog extends Animal{}
public class Test{
    public static void main(String[] args){
        Animal anl = method();
        Object anl = method();
        Dog d = (Dog)method();
    }

    // 参数为父类类型,就可以接收任意子类对象,或者该父类类型的对象
    // 返回值类型为父类类型,就可以返回任意子类对象,或者该父类类型的对象
    public static Animal method(){
        return new Dog();
        // return new Animal();
    }
}
```

引用类型转换

向上转型

- 概述: 子类向父类类型转换的过程
- eg: `Animal anl = new Dog();`

向下转型

- 概述: 父类类型向子类类型转换的过程
- eg: `Dog d = (Dog)anl;`
- 注意: 父类类型的变量指向的对象一定要是左边子类类型的对象

instanceof关键字

- 格式:

```
对象名 instanceof 数据类型
如果对象名指向的对象的类型是属于后面的数据类型,那么就返回true
如果对象名指向的对象的类型不是属于后面的数据类型,那么就返回false
```

- 案例:

```
public abstract class Animal{
    public abstract void eat();
}
public class Dog extends Animal{
    @Override
    public void eat(){
        System.out.println("狗吃骨头...");
    }

    public void lookHome(){
        System.out.println("看家....");
    }

}
public class Cat extends Animal{}

public class Test{
    public static void main(String[] args){
        method(new Dog());
        System.out.pritnln("--------")
        method(new Cat());
    }

    // 参数为父类类型,就可以接收任意子类对象,或者该父类类型的对象
    public static void method(Animal anl){
        anl.eat();
        // 向下转型
        if(anl instanceof Dog){
            Dog d = (Dog)anl;
            d.lookHome();
        }

    }

}
```

接口

定义格式

- 格式:

```
public interface 接口名{
    // 常量------>使用public static final修饰,这3个修饰符可以省略(jdk7及其以前)
    // 抽象方法--->使用public abstract修饰,这2个修饰符可以省略(jdk7及其以前)
    // 默认方法--->使用public default修饰,public可以省略,default不可以省略(jdk8)
    // 静态方法--->使用public static修饰,public可以省略,static不可以省略(jdk8)
    // 私有方法--->使用private修饰,private不可以省略  (jdk9及其以后)
}
```

- 案例:

```
package com.itheima.demo2_定义接口;

/**
 * @Author: pengzhilin
 * @Date: 2021/4/23 10:57
 */
public interface A {
    // 常量------>使用public static final修饰,这3个修饰符可以省略
    public static final int NUM = 10;

    // 抽象方法--->使用public abstract修饰,这2个修饰符可以省略
    public abstract void method1();

    // 默认方法--->使用public default修饰,public可以省略,default不可以省略
    public default void method2(){
        System.out.println("method2默认方法...");
    }

    // 静态方法--->使用public static修饰,public可以省略,static不可以省略
    public static void method3(){
        System.out.println("method3静态方法...");
    }

    // 私有方法--->使用private修饰,private不可以省略
    private void method4(){
        System.out.println("method4私有方法");
    }

    private static void method5(){
        System.out.println("method5私有静态方法");
    }

}
```

实现接口

- 单实现:

```
public class 实现类名 implements 接口名{

}
```

- 多实现:

```
public class 实现类名 implements 接口名1,接口名2,接口名3,....{

}
```

- 先继承后使用

```
public class 实现类名 extends 父类名 implements 接口名1,接口名2,接口名3,....{

}
```

- 如果要实现的接口中有抽象方法,实现类必须全部重写,否则实现类也得是个抽象类

接口中成员的访问特点

- 特点

```
常量: 一般供接口名直接访问
抽象方法:就是用来供实现类重写的
默认方法:可以供实现类继承直接使用,或者重写
静态方法: 只供接口名直接调用
私有方法: 只能在接口内部的其他方法中调用(默认方法,静态方法)
```

- 案例

```
/**
 * @Author: pengzhilin
 * @Date: 2021/4/23 11:26
 */
public class Test {
    public static void main(String[] args) {
        // 访问常量
        System.out.println(A.NUM);
        System.out.println(ImpA.NUM);

        // 访问抽象方法
        // 创建实现类对象
        ImpA impA = new ImpA();
        impA.method1();

        // 访问默认方法
        impA.method2();

        // 访问静态方法
        A.method3();
    }
}
```

接口和接口之间的关系

- 单继承

```
public interface A{}
public interface B extends A{}
```

- 多继承

```
public interface A{}
public interface B{}
public interface C extends A,B{}
```

- 多层继承

```
public interface A{}
public interface B extends A{}
public interface C extends B{}
```

集合

- 集合继承关系和特点:

```
单列集合:以单个单个元素进行存储数据
    List集合(接口): 元素可重复,元素有索引
        ArrayList类: 底层数据结构是数组,查询快,增删慢
        LinkedList类:底层数据结构是链表,查询慢,增删快

    Set集合(接口):  元素不可重复,元素无索引
        HashSet类:底层数据结构是哈希表结构,可以保证元素唯一
        LinkedHashSet类:底层数据结构是链表+哈希表,由哈希表保证元素唯一,由链表保证元素存取顺序一致
        TreeSet类:底层数据结构是红黑树,可以对元素进行排序

双列集合:以键值对的形式进行存储数据
    Map集合(接口): 键唯一,值可以重复,如果键重复了,值会被覆盖,根据键找值
        HashMap类:底层数据结构是哈希表结构,可以保证键唯一
        LinkedHashMap类:底层数据结构是链表+哈希表,由哈希表保证键唯一,由链表保证键值对存取顺序一致
        TreeMap类:底层数据结构是红黑树,可以对键进行排序
```

- 集合的api:
  - **Collection接口的api**
  - **List接口的api**
  - LinkedList类的api
  - **Map接口的api**
  - **Collections工具类的api**
- HashSet集合保证元素唯一的原理:

```
1.计算要存储的元素的哈希值
2.判断该哈希值对应的位置上是否有元素
3.如果该哈希值对应的位置上没有元素,就直接存储
4.如果该哈希值对应的位置上有元素,就产生了哈希冲突
5.产生了哈希冲突,就得调用元素的equals()方法与该位置上的所有元素进行一一比较:
  如果比较完之后,没有一个元素与该元素相等,就直接存储
  如果比较完之后,有任意一个元素与该元素现代,就不存储
```

- 注意: 如果元素是自定义类型,保证元素唯一,需要重写equals and hashCode方法

IO流

- 分类:

```
字节流: 以字节为基本单位进行读写数据
    字节输入流: 顶层父类是InputStream抽象类
        FileInputStream流: 读字节数据-->read()或者read(byte[] b)
        BufferedInputStream流: 读字节数据-->read()或者read(byte[] b)
        ObjectInputStream流: 特有的读对象的方法-->readObject()

    字节输出流: 顶层父类是OutputStream抽象类
        FileOutputStream流: 写字节数据-->write(int b)或者write(byte[] b,int
off,int len)
        BufferedOutputStream流: 写字节数据->write(int b)或者write(byte[] b,int
off,int len)
        ObjectOutputStream流: 特有的写对象的方法-->writeObject()
        PrintStream流: 特有的打印指定类型数据的方法-->println(任意类型的数据)或者
print(任意类型的数据)

字符流: 以字符为基本单位进行读写数据
    字符输入流: 顶层父类是Reader抽象类
        FileReader流: 读字符数据-->read()或者read(char[] chs)
        BufferedReader流: 特有的方法-->readLine()
        InputStreamReader流:  1.把字节输入流转换为字符输入流    2.指定编码读数据

    字符输出流: 顶层父类是Writer抽象类
        FileWrtier流:写字符数据--->write(int c)或者write(char[] chs,int off,int
len)
        BufferedWriter流: 特有的换行方法--->newLine()
        OutputStreamWriter流: 1.把字节输出流转换为字符输出流    2.指定编码写数据
```

- IO流使用步骤:

```
固定步骤:
1.创建输入流对象,关联数据源文件路径
2.创建输出流对象,关联目的地文件路径
3.定义变量,用来存储读取到的数据
4.循环读数据
5.在循环中,写数据
6.关闭流,释放资源
```

属性集

- 相关api:

```
public Properties() :创建一个空的属性列表。
public void load(InputStream inStream)： 从字节输入流中读取键值对。
public void load(Reader reader) 从字符输入流中读取键值对。

public Set<String> stringPropertyNames() ：所有键的名称的集合。
public String getProperty(String key) ：使用此属性列表中指定的键搜索属性值。
```

- 案例:

```
driverClass=com.mysql.jdbc.Driver
password=123456
jdbcUrl=jdbc\:mysql\://localhost\:3306/web17
username=root
```

```
package com.itheima.utils;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.util.Properties;

/**
 * @Author: pengzhilin
 * @Date: 2021/4/23 15:11
 */
public class JDBCUtils {
    public static String driverClass = null;
    public static String jdbcUrl = null;
    public static String username = null;
    public static String password = null;
    static {
        try {
            // 1.创建Properties对象
            Properties pro = new Properties();

            // 2.调用load方法加载配置文件
            InputStream is =
Class.forName("com.itheima.utils.JDBCUtils").getClassLoader().getResourceAsS
tream("db.properties");
            pro.load(is);

            // 3.获取数据
            driverClass = pro.getProperty("driverClass");
            jdbcUrl = pro.getProperty("jdbcUrl");
            username = pro.getProperty("username");
            password = pro.getProperty("password");

            // 4.修改pro对象中password键对应的值
            pro.setProperty("password","123456");

            // 5.然后把pro对象写回到配置文件中
            pro.store(new
FileOutputStream("day16\\src\\db.properties"),"itheima");
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {

        System.out.println(driverClass);
        System.out.println(jdbcUrl);
        System.out.println(username);
        System.out.println(password);

    }
}
```

反射

通过反射获取类的成员方法

```
Class类中与Method相关的方法
* Method getDeclaredMethod(String name,Class... args);----->推荐
    * 根据方法名和参数类型获得对应的构造方法对象，包括public、protected、(默认)、private的
      参数1:要获取的方法的方法名
      参数2:要获取的方法的形参类型的Class对象
* Method[] getDeclaredMethods();----->推荐
    * 获得类中的所有成员方法对象，返回数组,只获得本类的，包括public、protected、(默认)、
private的
```

通过反射执行成员方法

```
Method对象常用方法
*  Object invoke(Object obj, Object... args)
    * 参数1:调用该方法的对象
    * 参数2:调用该法时传递的实际参数
      返回值:该方法执行完毕后的返回值

*  void setAccessible(true)
    设置"暴力访问"--是否取消权限检查，true取消权限检查，false表示不取消
```

示例代码

```
public class Person {

    public void show1(){
        System.out.println("无参数无返回值show1...");
    }

    public double show2(int num,String str){
        System.out.println("有参数有返回值show2...num:"+num+",str:"+str);
        return 3.14;
    }
```

```
    private double show1(int num,String str){
        System.out.println("有参数有返回值show1...num:"+num+",str:"+str);
        return 4.2;
    }
}
```

```
public class Test {
    public static void main(String[] args)throws Exception {
        // 1.获取Person类的字节码对象
        Class<Person> c = Person.class;

        // 2.获取成员方法对象
        Method m1 = c.getDeclaredMethod("show1");
        Method m2 = c.getDeclaredMethod("show1",int.class,String.class);
        Method m3 = c.getDeclaredMethod("show2", int.class, String.class);

        // 3.执行成员方法
        // 通过反射得到Person类的对象
        Person p = c.getDeclaredConstructor().newInstance();
        m1.invoke(p);

        // 取消m2表示的方法的权限检查
        m2.setAccessible(true);
        System.out.println(m2.invoke(p, 100, "itheima"));// 4.2

        System.out.println(m3.invoke(p, 200, "itcast"));// 3.14
    }
}
```

jdk8新特性

Lambda,Stream流,方法引用

- Lambda表达式

  ```
  格式: (参数列表)->{代码块}
  前提: 函数式接口
  使用套路:
      1.分析是否可以使用Lambda表达式
      2.如果可以使用,就直接写上()->{}
      3.填充小括号中的内容-->和函数式接口中抽象方法的形参列表一致
      4.填充大括号中的内容-->实现函数式接口抽象方法的方法体一致
  省略规则:
      1.小括号中参数类型可以省略
      2.小括号中如果只有一条语句,那么小括号也可以省略
      3.大括号中如果只有一条语句,那么大括号,分号,return都可以省略(一起省略)
  ```

- Stream流
  - 使用步骤: 获取流--->操作流---->收集结果
  - Stream流api:

        - forEach
        - count
        - collect
        - filter
        - limit
        - skip
        - map
        - concat

    - 案例:

```
public class Test1_Stream流和Lambda表达式 {
    public static void main(String[] args) {
        // 1.获取流
        Stream<String> stream1 = Stream.of("张三丰", "张翠山", "金毛狮王",
"张无忌");
        Stream<String> stream2 = Stream.of("110", "120", "119", "114");

        // 2.操作流--->过滤出姓张的元素,并取前2个,打印输出
        //stream1.filter(t->t.startsWith("张")).limit(2).forEach(t->
System.out.println(t));

        // 3.操作流--->转换Integer类型,并跳过前2个,打印输出
        //stream2.map(t->Integer.parseInt(t)).skip(2).forEach(t->
System.out.println(t+1));


        // 4.操作流--->把姓名转换为Person对象,打印输出
        //stream1.map(t->new Person(t)).forEach(t->
System.out.println(t));

        // 5.操作流--->把姓名转换为姓名对应的字符长度,打印输出
        stream1.map(t->t.length()).forEach(t-> System.out.println(t));
    }
}
```

- 方法引用

```
方法引用:
1.判断是否可以使用方法引用替换Lambda表达式--->使用场景
2.如果可以使用,那么就得确定引入方法的类型
3.确定引入的方法的类型之后,根据引入格式引入该方法即可
方法的类型:
    静态方法: 类名::方法名
    构造方法: 类名::new
    无参数成员方法: 类名::方法名
    有参数的成员方法: 对象名::方法名
```

```java
/*
    一个java文件可以定义多个类,但该文件中只能有一个public修饰的类,并且这个public修饰的类的类名和文件名一致
 */
 class Student{
    private String name;

    public Student(String name) {
        this.name = name;
    }

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

public class Test2_Stream流和方法引用{
    public static void main(String[] args) {
        // 1.获取流
        Stream<String> stream1 = Stream.of("张三丰", "张翠山", "金毛狮王", "张无忌");
        Stream<String> stream2 = Stream.of("110", "120", "119", "114");

        // 2.操作流--->过滤出姓张的元素,并取前2个,打印输出
        //stream1.filter(t->t.startsWith("张")).limit(2).forEach(t->System.out.println(t));
        //stream1.filter(t->t.startsWith("张")).limit(2).forEach(System.out::println);

        // 3.操作流--->转换Integer类型,并跳过前2个,打印输出
        //stream2.map(t->Integer.parseInt(t)).skip(2).forEach(t->System.out.println(t+1));
        //stream2.map(Integer::parseInt).skip(2).forEach(t->System.out.println(t+1));



        // 4.操作流--->把姓名转换为Student对象,打印输出
        //stream1.map(t->new Student(t)).forEach(t-> System.out.println(t));
        //stream1.map(Student::new).forEach(System.out::println);


        // 5.操作流--->把姓名转换为姓名对应的字符长度,打印输出
        //stream1.map(t->t.length()).forEach(t-> System.out.println(t));
        stream1.map(String::length).forEach(System.out::println);

    }
}
```

线程安全

线程创建和启动

- 线程创建:
  - 继承方式:
    - 创建一个线程子类继承Thread类
    - 在线程子类中重写run方法,把线程需要执行的任务代码放入run方法中
    - 创建线程子类对象
    - 调用start()方法启动线程,执行任务
  - 实现方式:
    - 创建一个实现类实现Runnable接口
    - 在实现类中重写run方法,把线程需要执行的任务代码放入run方法中
    - 创建Thread线程对象,并传入任务实现类对象
    - 调用start()方法启动线程,执行任务
  - 线程的调度: 抢占式
- 线程的状态:
  - 新建
  - 可运行
  - 锁阻塞
  - 无限等待
  - 计时等待
  - 被终止
- 等待唤醒机制:
  - 如何实现等待唤醒机制程序:
    - 使用锁对象调用wait()方法进入无限等待
    - 使用锁对象调用notify()\notifyAll()方法唤醒无限等待线程
    - 调用wait(),notify()\notifyAll()方法的锁对象要一致,否则无法实现
    - 写代码:
      - 分析什么条件下进入无限等待,什么情况下进入唤醒状态
      - eg:
        - 包子铺线程: 如果flag的值为true,就进入无限等待,如果flag为false,就进入唤醒
        - 吃货线程: 如果flag的值为flase,就进入无限等待,如果flag为true,就进入唤醒
  - 如何分析等待唤醒机制程序:
    - 线程的调度是抢占式
    - 线程进入无限等待之后,不会争夺cpu,会释放锁对象
    - 线程从无限等待状态下被唤醒之后,会从进入无限等待的位置继续往下执行
    -

可见性问题演示

- **概述: 一个线程没有看见另一个线程对共享变量的修改**
- 例如下面的程序，先启动一个线程，在线程中将一个变量的值更改，而主线程却一直无法获得此变量的新值。

> 1. 线程类：

```
```

```
public class MyThread extends Thread {

    static boolean flag = false;// 主和子线程共享变量

    @Override
    public void run() {

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 把flag的值改为true
        flag = true;
        System.out.println("修改后flag的值为:"+flag);

    }
}
```

1. 测试类:

```
public class Test {
    public static void main(String[] args) {
        /*
            多线程的安全性问题-可见性:
                一个线程没有看见另一个线程对共享变量的修改
         */
        // 创建子线程并启动
        MyThread mt = new MyThread();
        mt.start();

        // 主线程
        while (true){
            if (MyThread.flag == true){
                System.out.println("死循环结束");
                break;
            }
        }
        /*
            分析后期望的结果:主线程一直死循环,当子线程把共享变量flag的值改为true,主线程就结束死循环
            实际结果: 主线程一直死循环,当子线程把共享变量flag的值改为true,主线程依然还是死循环
            原因: 子线程对共享变量flag值的改变,对主线程不可见
            解决办法: 使用volatile关键字,当变量被修饰为volatile时，会迫使线程每次使用此变量，都会去主内存获取，保证其可见性
         */
    }
}
```

- 原因:
- JMM内存模型(Java Memory Model)描述了Java程序中**各种变量(线程共享变量)的访问规则**，以及在JVM中将变量存储到内存和从内存中读取变量这样的底层细节。

- 简而言之: 就是所有共享变量都是存在主内存中的,线程在执行的时候,有单独的工作内存,会把共享变量拷贝一份到线程的单独工作内存中,并且对变量所有的操作,都是在单独的工作内存中完成的,不会直接读写主内存中的变量值

可见性问题解决

- 解决办法: **使用volatile关键字,当变量被修饰为volatile时，会迫使线程每次使用此变量，都会去主内存获取，保证其可见性**
- 代码:

```
public class MyThread extends Thread {

    volatile static boolean flag = false;// 主和子线程共享变量

    @Override
    public void run() {

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // 把flag的值改为true
        flag = true;
        System.out.println("修改后flag的值为:"+flag);

    }
}
```

有序性问题演示和解决

有序性问题演示

- 有些时候“编译器”在编译代码时，会对代码进行“重排”，例如:

```
int a = 10;    //1
int b = 20;    //2
int c = a + b;  //3
```

- 单线程：第一行和第二行可能会被“重排”：可能先编译第二行，再编译第一行，总之在执行第三行之前，会将1,2编译完毕。1和2先编译谁，不影响第三行的结果。
- 但在“多线程”情况下，代码重排，可能会对另一个线程访问的结果产生影响：

多线程环境下，我们通常不希望对一些代码进行重排的！！

有序性问题解决

- **使用volatile修饰共享变量,禁止编译器重排**

原子性问题演示

- 概述：所谓的原子性是指在一次操作或者多次操作中，要么所有的操作全部都得到了执行并且不会受到任何因素的干扰而中断，要么所有的操作都不执行，**多个操作是一个不可以分割的整体。**
- 请看以下示例：
  - 一条子线程和一条主线程都对共享变量a进行++操作,每条线程对a++操作100000次

> 1.制作线程类

```
public class MyThread extends Thread{
    static int a = 0;

    @Override
    public void run() {
        // 子线程对a进行自增10万次
        for (int i = 0; i < 100000; i++) {
            a++;
        }
        System.out.println("子线程执行完毕");
    }
}
```

```
public class Test {
      public static void main(String[] args) throws Exception{
          // 创建并启动子线程
          new MyThread().start();

          // 主线程对a进行自增10万次
```

```
        for (int i = 0; i < 100000; i++) {
            MyThread.a++;
        }

        // 为了保证子线程和主线程都执行完毕
        Thread.sleep(3000);

        // 打印最终共享变量a的值(子线程,主线程对a的操作都执行完毕了)
        System.out.println("最终:"+MyThread.a);
        /*
            分析后期望的结果: 20万
            实际的结果: 小于或者等于20万
            原因: 主线程和子线程同时对a进行自增,产生了覆盖的效果
         */
    }
}
```

原因：两个线程对共享变量的操作产生覆盖的效果

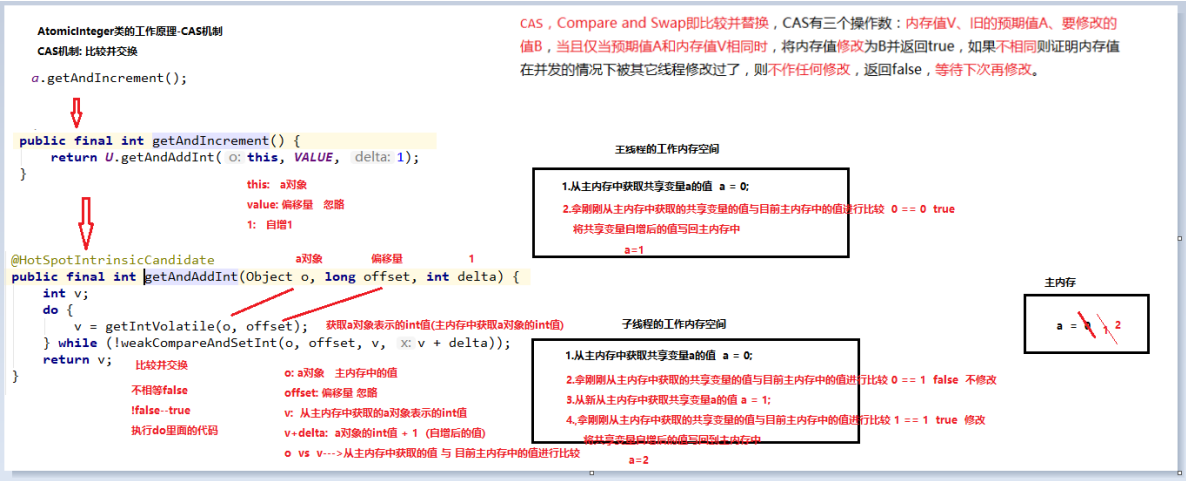
原子性问题解决

- 解决办法: **加锁,或者使用原子类**
- 代码:

```
public class MyThread extends Thread{
    //static int a = 0;
    static AtomicInteger a = new AtomicInteger(0);

    @Override
    public void run() {
        // 子线程对a进行自增10万次
        for (int i = 0; i < 100000; i++) {
            /*synchronized ("suo"){
                a++;
            }*/

            a.getAndIncrement();
        }
        System.out.println("子线程执行完毕");
    }
}


public class Test {
    public static void main(String[] args) throws Exception {
        // 创建并启动子线程
        new MyThread().start();

        // 主线程对a进行自增10万次
        for (int i = 0; i < 100000; i++) {
            /*synchronized ("suo"){
                MyThread.a++;
            }*/
            MyThread.a.getAndIncrement();
        }
```

```
            // 为了保证子线程和主线程都执行完毕
            Thread.sleep(3000);

            // 打印最终共享变量a的值(子线程,主线程对a的操作都执行完毕了)
            System.out.println("最终:" + MyThread.a.get());
            /*
                解决办法: 加锁,或者原子类
             */
        }
}
```

AtomicInteger类的工作原理-CAS机制



同步代码块

- 格式:

```
synchronized(锁对象){

}
```

- 锁对象
  - 语法: 可以是任意类对象
  - 注意: 如果多条线程想要实现同步,那么这多条线程的锁对象必须一致(相同)
- 同步方法:
- 格式: 方法的返回值类型前面加上synchronized,其余都不变
- 锁对象:
  - 非静态成员方法: 锁对象就是this
  - 静态成员方法: 该方法所在类的字节码对象,类名.class

Lock锁

- `public void lock();` 加锁
- `public void unlock();` 释放锁