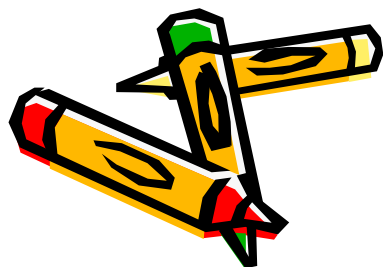
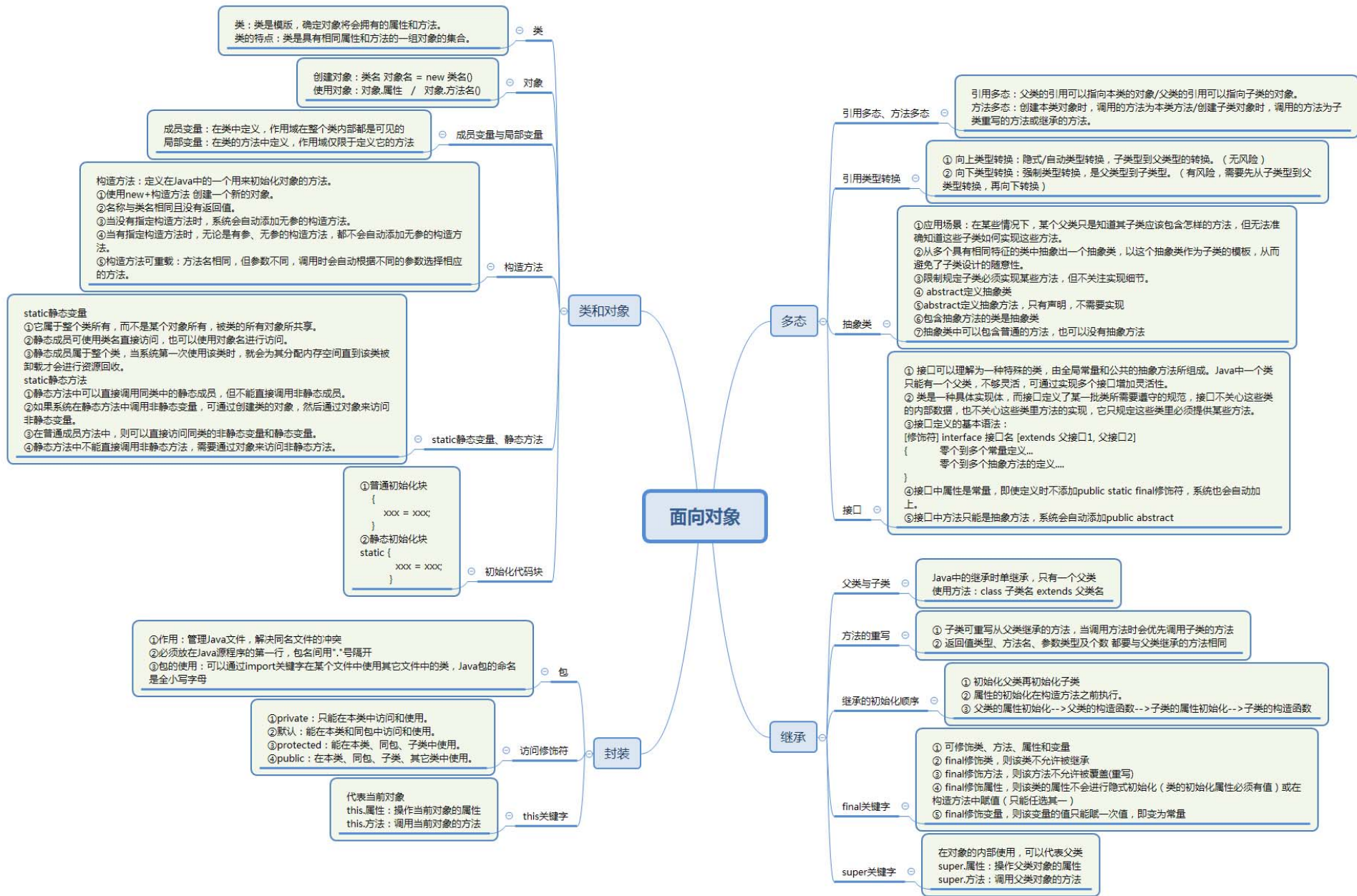


第3章 类的封装、继承和多态



- 3.1 类和对象
- 3.2 类的封装性
- 3.3 类的继承性
- 3.4 类的多态性
- 3.5 类的抽象性
- 3.6 数组
- 3.7 字符串







3.1 类和对象（组织代码，封装数据）

面向对象和面向过程的本质区别-程序设计语言进化史：

语言的进化发展跟生物的进化发展其实是一回事，都是”物以类聚”。相近的感光细胞聚到一起变成了我们的眼睛，相近的嗅觉细胞聚到一起变成了我们的鼻子。

语句多了，我们将完成同样功能的相近的语句，聚到了一块儿，便于我们使用。于是，方法出现了！

变量多了，我们将功能相近的变量组在一起，聚到一起归类，便于我们调用。于是，结构体出现了！

再后来，方法多了，变量多了！结构体不够用了！我们就将功能相近的变量和方法聚到了一起，于是类和对象出现了！



3.1 类和对象（组织代码，封装数据）

面向过程的思维模式：

面向过程的思维模式是简单的线性思维，思考问题首先陷入第一步做什么、第二步做什么的细节中。这种思维模式适合处理简单的事情。但如果面对复杂特别是需要很多人来协作完成的事情，单纯采用这种思维模式就行不通！！！！

面向对象的思维模式：

面向对象的思维模式说白了就是分类思维模式。思考问题首先会解决问题需要哪些分类，然后对这些分类进行单独思考。最后，才对某个分类下的细节进行面向过程的思索。这样就可以形成很好的协作分工。比如：设计师分了**100**个类，然后将**100**个类交给了**100**个人分别进行详细设计和编码！显然，面向对象适合处理复杂的问题，适合处理需要多人协作的问题！如果一个问题需要多人协作一起解决，那么你一定要用面向对象的方式来思考！

对于描述复杂的事物，为从宏观上把握、从整体上合理分析，需要使用面向对象的思路来分析整个系统。但是，具体到微观操作，仍然需要面向过程的思路去处理。



3.1 类和对象（组织代码，封装数据）

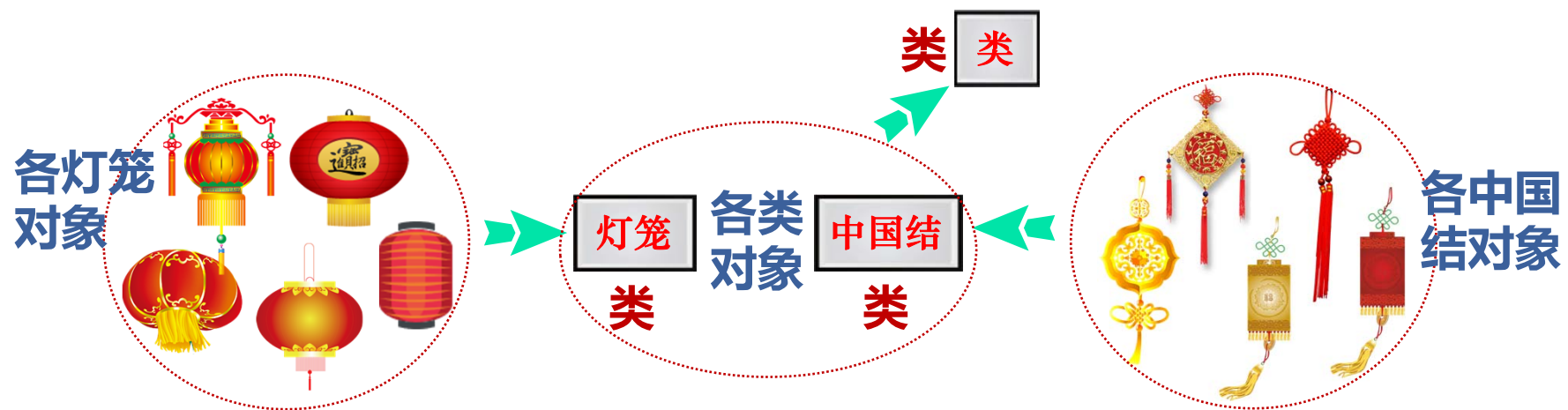
面向对象编程的本质：以类的方式组织代码！以对象的方式封装数据！

类（class）是既包括数据又包括作用于数据的一组操作的封装体。类可以看成是一类相似对象的模板！类具有封装性、继承性、多态性和抽象性。

对象（object）是类的具体**实例（instance）**。**Java中，万事万物皆对象！！！Java**中通过引用操作对象！引用就是指向对象的内存中地址！！

从**认识论角度**考虑是先有对象后有类。对象，是具体的事物。类，是抽象的，是对对象的抽象。从**代码运行角度**考虑是先有类后有对象。类是对象的模板。

3.1 类和对象 (组织代码, 封装数据)





3.1 类的属性/Java中的变量

局部变量、实例变量、类变量（在类中，用**static**声明的成员变量为静态成员变量，它为该类的公用变量、属于类，被该类的所有实例共享，在类被载入时被显示初始化；对于该类的所有对象来说，**static**成员变量只有一份；可以使用“对象.类属性”来调用，不过，一般都是用”类名.类属性”；**static**变量置于方法区中）。局部变量使用前必须先赋值、而实例变量则有缺省初值（数值：**0**，**0.0**；字符**char**: **\u0000**；布尔类型**boolean**: **false**；所有引用类型: **null**）。方法内只能使用自己声明的局部变量、参数、实例变量、类变量，不能使用其它方法内的局部变量。两个方法内的同名局部变量互不影响，哪怕它们同名。



3.1 类的方法

方法是类和对象动态行为特征的抽象。方法很类似于面向过程中的函数。面向过程中，函数是最基本单位，整个程序有一个一个函数调用组成；面向对象中，整个程序的基本单位是类，方法是从属于类或对象的。

方法定义格式：

```
[修饰符] 方法返回值类型 方法名(形参列表) {  
    // n条语句  
}
```

修饰符可省略，可是**public,protected,private,static,final,abstract**等；类型可以是基本类型和引用类型或**void**。



3.1 类的方法

方法：

- **设计方法的原则**：方法的本意是功能块，就是实现某个功能的语句块的集合。我们设计方法的时候，最好保持方法的原子性，就是一个方法只完成**1**个功能，这样利于我们后期的扩展。
- **形式参数**：在方法被调用时用于接收外界输入的数据；**实参**：调用方法时实际传给方法的数据；**返回值**：方法在执行完毕后返还给调用它的环境的数据；**返回值类型**：事先约定的返回值的数据类型，如无返回值，必须给出返回值类型**void**。**return语句**终止方法的运行并指定要返回的数据。
- **Java**语言中使用下述形式调用方法：对象名.方法名(实参列表)。
- 实参的数目、数据类型和次序必须和所调用方法声明的形参列表匹配。
- **Java**中进行方法调用中传递参数时，**遵循值传递的原则**：基本类型传递的是该数据值本身。引用类型传递的是对对象的引用，而不是对象本身。
- **Primitive → passing by value; Objects → passing by object(reference)**



3.1.1 类

1. 声明类

类声明

{

成员变量的声明;

成员方法的声明及实现;

}

[修饰符] **class** 类<泛型> [**extends** 父类]
[**implements** 接口列表]



3.1.1 类

2. 声明成员变量和成员方法

[修饰符] 返回值类型 **方法**(**[参数列表]**)
[throws 异常类]

{

语句序列;

[return [返回值]];

}

3. 成员方法重载



3.1.2 对象

1. 声明对象

类 对象;

2. 创建实例

对象 = **new** 类的构造方法([参数列表])

3. 引用对象的成员变量和调用对象方法

对象.成员变量

对象.成员方法([参数列表])

Java语言自动释放对象。



3.1 Java程序内存分析

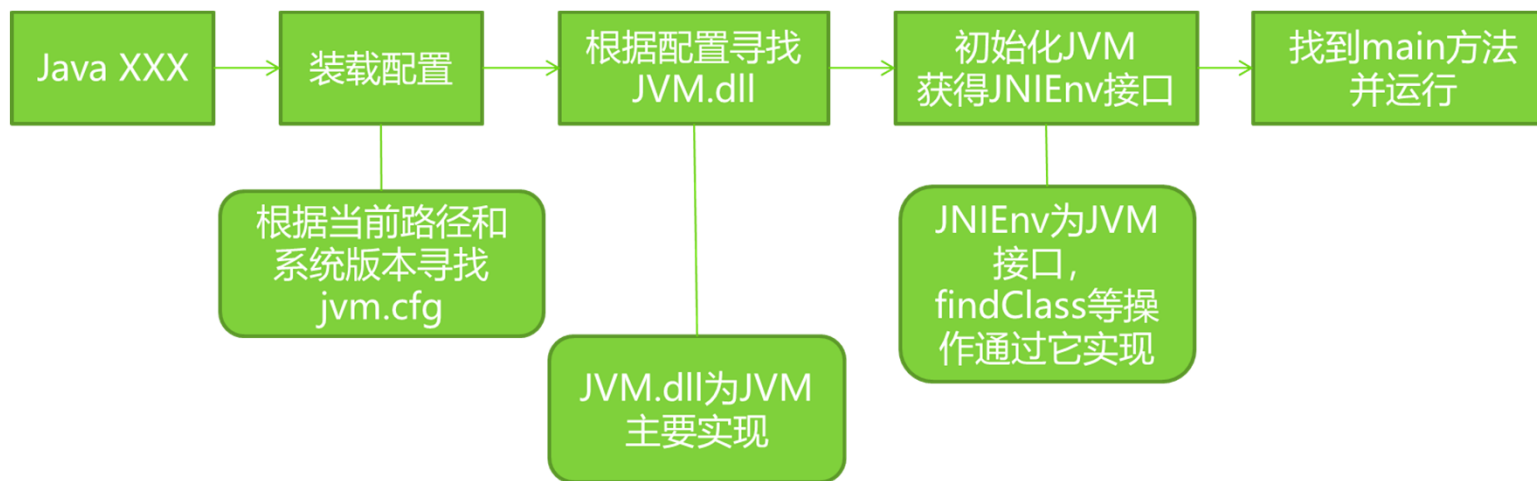
- **Hexadecimal**: 十六进制表示地址。内存分一个个单元，每个单元**1byte**（**8 bits**），都是有地址（十六进制表示）的。所以如果你**16G**的内存条，就是**160**亿个位置，需要**160**亿个地址，如果内存够大，**FFFF....**的个数会越多。
- **栈stack**: 由系统自动分配连续的空间，后进先出，放置局部变量（**local variables/parameters→stack**）。速度快。
- **堆heap**: 不连续，放置**new**出来的对象（**Dynamic variables→heap**），分配灵活，速度慢。
- **方法区**: 也是堆，存放类的代码信息、**static**变量、**static**方法、常量池（字符串常量）。（**static variables/constant→special**）。用来存放程序中不变或唯一的内容。
- 方法区和堆依次存放于内存的低区域（这里的地址数值比较小），堆的地址从小往大分配；栈存放于内存的高区域（地址数值大，从**FFF...**开始），栈的地址从大往小分配。当互相重叠时，就互相修改数据，程序崩溃。



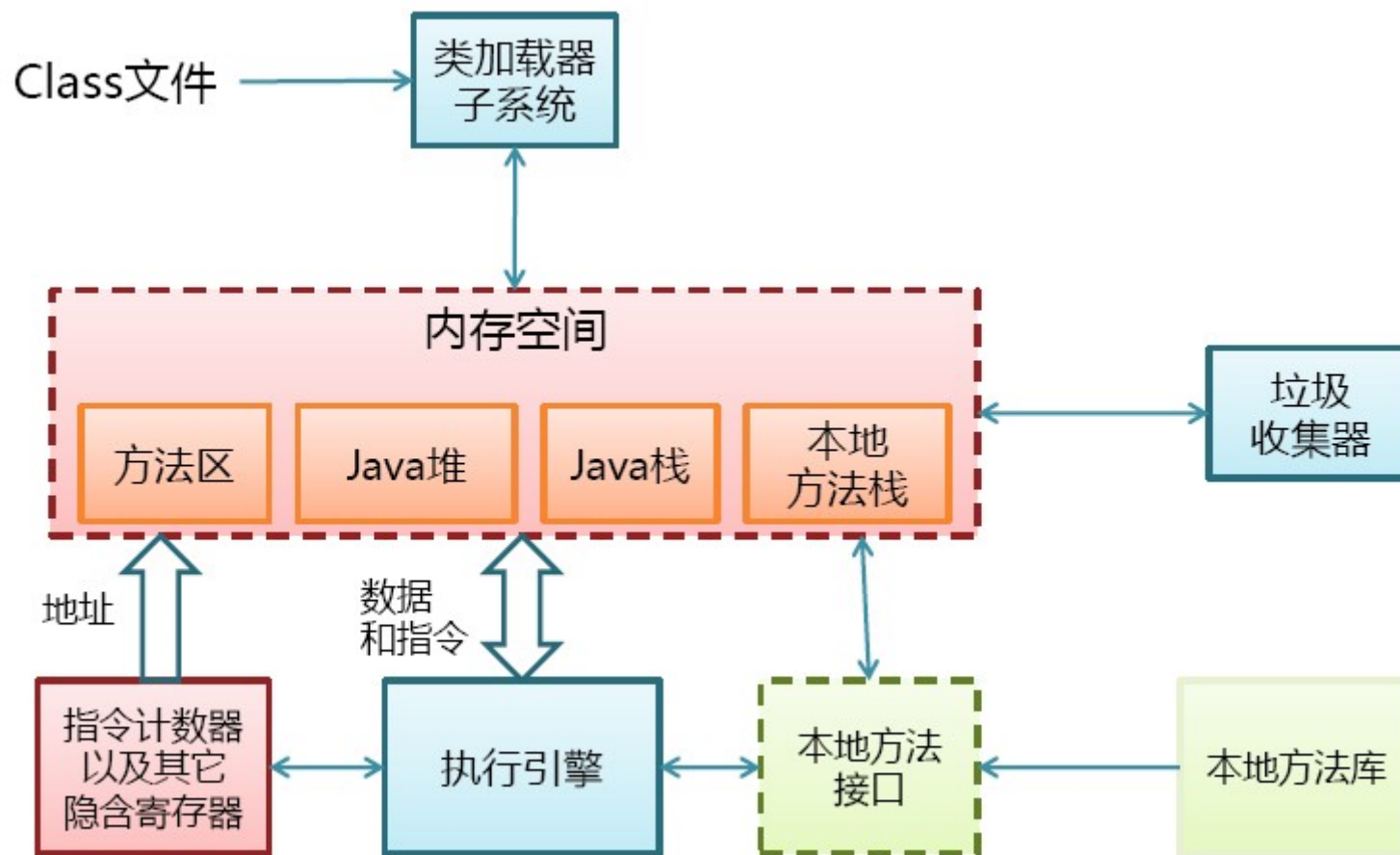
3.1 垃圾回收机制 (Garbage Collection)

- 使用**new**关键字创建对象时，进行对象空间的分配；
- 当对象没有引用指向时（将对象赋值**null**即可），垃圾回收器将负责回收所有不可达对象的内存空间；
- **java**程序员无权调用垃圾回收器；
- 程序员可以通过**system.gc()**通知**GC**运行，但是**java**规范并不能保证立刻运行
- **finalize**方法，是**Java**提供给程序员用来释放对象或资源的方法，一般用不上

3.1 JVM启动流程



3.1 JVM基本结构





3.1 JVM基本结构

PC寄存器

- ① 每个线程拥有一个**PC**寄存器
- ② 在线程创建时创建
- ③ 指向下一条指令的地址
- ④ 执行本地方法时，**PC**的值为**undefined**



3.1 JVM基本结构

方法区

① 保存装载的类信息

- 运行时常量池（存放字面常量如文本字符串、**final**常量值；以及字段、方法等的符号引用等）
- 字段，方法信息
- 方法字节码

② 通常和永久区(**Perm**)关联在一起



3.1 JVM基本结构

Java堆

- ① 和程序开发密切相关
- ② 应用系统对象都保存在**Java堆**中
- ③ 所有线程共享**Java堆**
- ④ 对分代**GC**来说，堆也是分代的
- ⑤ **GC**的主要工作区间



复制算法



3.1 JVM基本结构

Java栈

- ① 线程私有
- ② 栈由一系列帧组成（因此**Java**栈也叫做帧栈）
- ③ 帧保存一个方法的局部变量、操作数栈、常量池指针
- ④ 每一次方法调用创建一个帧，并压栈



3.1 JVM基本结构

Java栈 – 局部变量表 包含参数和局部变量

```
public class StackDemo {  
  
    public static int runStatic(int i,long l,float f,Object o ,byte b){  
        return 0;  
    }  
  
    public int runInstance(char c,short s,boolean b){  
        return 0;  
    }  
}
```

0	int	int i
1	long	long l
3	float	float f
4	reference	Object o
5	int	byte b

0	reference	this
1	int	char c
2	int	short s
3	int	boolean b

3.1 JVM基本结构

Java栈 – 函数调用组成帧栈

```
public static int runStatic(int i,long l,float f,Object o ,byte b){  
    return runStatic(i,l,f,o,b);  
}
```

这是一个帧

省略：操作数栈 返回地址等

0	int	int i
1	long	long l
3	float	float f
4	reference	Object o
5	int	byte b

0	int	int i
1	long	long l
3	float	float f
4	reference	Object o
5	int	byte b

0	int	int i
1	long	long l
3	float	float f
4	reference	Object o
5	int	byte b

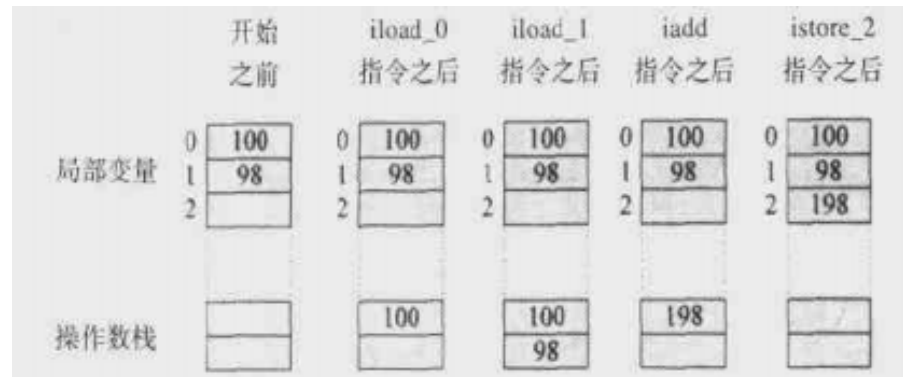
3.1 JVM基本结构

Java栈 – 操作数栈

Java没有寄存器，所有参数传递使用操作数栈

```
public static int add(int a,int b){  
    int c=0;  
    c=a+b;  
    return c;  
}
```

```
0: iconst_0 // 0压栈  
1: istore_2 // 弹出int, 存放于局部变量2  
2: iload_0 // 把局部变量0压栈  
3: iload_1 // 局部变量1压栈  
4: iadd     //弹出2个变量, 求和, 结果压栈  
5: istore_2 //弹出结果, 放于局部变量2  
6: iload_2 //局部变量2压栈  
7: ireturn  //返回
```



3.1 JVM基本结构

Java栈 – 栈上分配

C++ 代码示例

```
class BcmBasicString{ ....}
```

```
public void method(){  
    BcmBasicString* str=new  
    BcmBasicString; .... delete str;  
}
```

堆上分配
每次需要清理空间

```
public void method(){  
    BcmBasicString str;  
    ....  
}
```

栈上分配
函数调用完成自动清理



3.1 JVM基本结构

Java栈 – 栈上分配

```
public class OnStackTest {  
    public static void alloc(){  
        byte[] b=new byte[2];  
        b[0]=1;  
    }  
    public static void main(String[] args) {  
        long b=System.currentTimeMillis();  
        for(int i=0;i<100000000;i++){  
            alloc();  
        }  
        long e=System.currentTimeMillis();  
        System.out.println(e-b);  
    }  
}
```

```
-server -Xmx10m -Xms10m  
-XX:+DoEscapeAnalysis -XX:+PrintGC
```

输出结果 5

```
-server -Xmx10m -Xms10m  
-XX:-DoEscapeAnalysis -XX:+PrintGC
```

```
.....  
[GC 3550K->478K(10240K), 0.0000977 secs]  
[GC 3550K->478K(10240K), 0.0001361 secs]  
[GC 3550K->478K(10240K), 0.0000963 secs]  
564
```



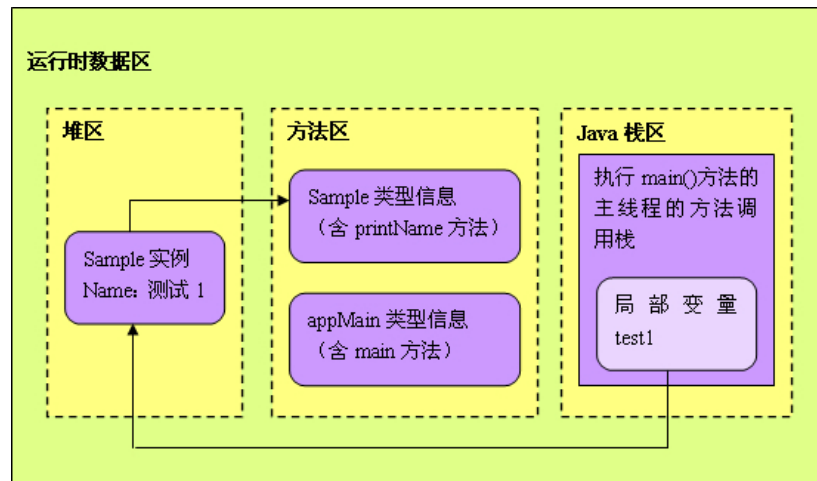
3.1 JVM基本结构

Java栈 – 栈上分配

- ① 小对象（一般几十个**bytes**），在没有逃逸的情况下，可以直接分配在栈上
- ② 直接分配在栈上，可以自动回收，减轻**GC**压力
- ③ 大对象或者逃逸对象无法栈上分配

3.1 JVM基本结构

栈、堆、方法区交互



```
public class AppMain
//运行时, jvm 把Appmain的信息都放入方法区
{
    public static void main(String[] args)
    //main 方法本身放入方法区。
    {
        Sample test1 = new Sample( " 测试1 " );
        //test1是引用, 所以放到栈区里, Sample是自定义对象应该放到堆里面
        Sample test2 = new Sample( " 测试2 " );

        test1.printName();
        test2.printName();
    }
}
```

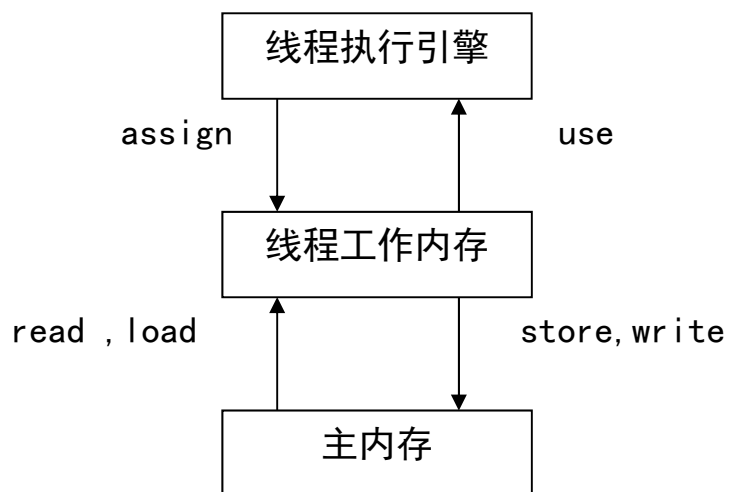
```
public class Sample
//运行时, jvm 把Sample的信息都放入方法区
{
    private name;
    //new Sample实例后, name 引用放入栈区里, name 对象放入堆里

    public Sample(String name)
    {
        this.name = name;
    }
    //print方法本身放入 方法区里。
    public void printName()
    {
        System.out.println(name); } }
```

3.1 JVM基本结构

内存模型

- ① 每一个线程有一个工作内存和主存独立
- ② 工作内存存放主存中变量的值的拷贝



当数据从主内存复制到工作存储时，必须出现两个动作：第一，由主内存执行的读（read）操作；第二，由工作内存执行的相应的load操作；当数据从工作内存拷贝到主内存时，也出现两个操作：第一个，由工作内存执行的存储（store）操作；第二，由主内存执行的相应的写（write）操作

每一个操作都是原子的，即执行期间不会被中断

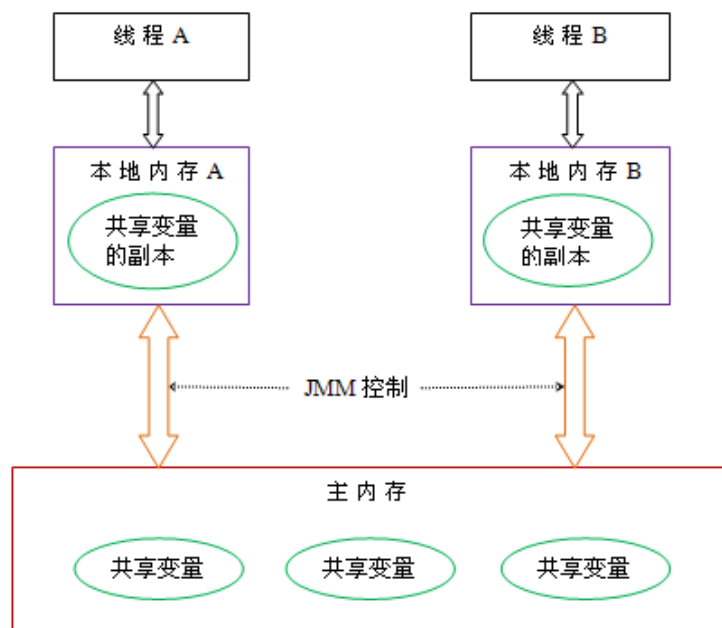
对于普通变量，一个线程中更新的值，不能马上反应在其他变量中

如果需要在其他线程中立即可见，需要使用 volatile 关键字

3.1 JVM基本结构

内存模型

- ① 每一个线程有一个工作内存和主存独立
- ② 工作内存存放主存中变量的值的拷贝





3.1 JVM基本结构

volatile

```
public class VolatileStopThread extends Thread{
    private volatile boolean stop = false;
    public void stopMe(){
        stop=true;
    }

    public void run(){
        int i=0;
        while(!stop){
            i++;
        }
        System.out.println("Stop thread");
    }

    public static void main(String args[]) throws
    InterruptedException{
        VolatileStopThread t=new VolatileStopThread();
        t.start();
        Thread.sleep(1000);
        t.stopMe();
        Thread.sleep(1000);
    }
}
```

没有volatile -server 运行 无法停止

volatile 不能代替锁
一般认为volatile 比锁性能好（不绝对）

选择使用volatile的条件是：
语义是否满足应用



3.1 JVM基本结构

1. 解释运行

- ① 解释执行以解释方式运行字节码
- ② 解释执行的意思是：读一句执行一句

2. 编译运行（**JIT**）

- ① 将字节码编译成机器码
- ② 直接执行机器码
- ③ 运行时编译
- ④ 编译后性能有数量级的提升



3.1 class装载验证流程

1. 加载
2. 链接
 - ① 验证
 - ② 准备
 - ③ 解析
3. 初始化



3.1 class装载验证流程

加载:

1. 装载类的第一个阶段
2. 取得类的二进制流
3. 转为方法区数据结构
4. 在**Java**堆中生成对应的**java.lang.Class**对象



3.1 class装载验证流程

链接 -> 验证

- ① 目的：保证**Class**流的格式是正确的
 - 文件格式的验证
 - 是否以**0xCAFEBA**BE开头
 - 版本号是否合理
 - 元数据验证
 - 是否有父类
 - 继承了**final**类?
 - 非抽象类实现了所有的抽象方法
 - 字节码验证 (很复杂)
 - 运行检查
 - 栈数据类型和操作码数据参数吻合
 - 跳转指令指定到合理的位置
 - 符号引用验证
 - 常量池中描述类是否存在
 - 访问的方法或字段是否存在且有足够的权限



3.1 class装载验证流程

链接 -> 准备

① 分配内存，并为类设置初始值（方法区中）


- **public static int v=1;**
- 在准备阶段中，**v**会被设置为**0**
- 在初始化的<**clinit**>中才会被设置为**1**
- 对于**static final**类型，在准备阶段就会被赋上正确的值
- **public static final int v=1;**




3.1 class装载验证流程

链接 -> 解析

① 符号引用替换为直接引用



字符串
引用对象不一定
被加载



指针或者地址偏
移量
引用对象一定在
内存



3.1 class装载验证流程

初始化:

1. 执行类构造器<clinit>
 - ① **static**变量 赋值语句
 - ② **static{}**语句
2. 子类的<clinit>调用前保证父类的<clinit>被调用
3. <clinit>是线程安全的



3.1 class装载验证流程

1. **class**装载器**ClassLoader**是一个抽象类
2. **ClassLoader**的实例将读入**Java**字节码将类装载到**JVM**中
3. **ClassLoader**可以定制，满足不同的字节码流获取方式
4. **ClassLoader**负责类装载过程中的加载阶段



3.1 class装载验证流程

ClassLoader的重要方法

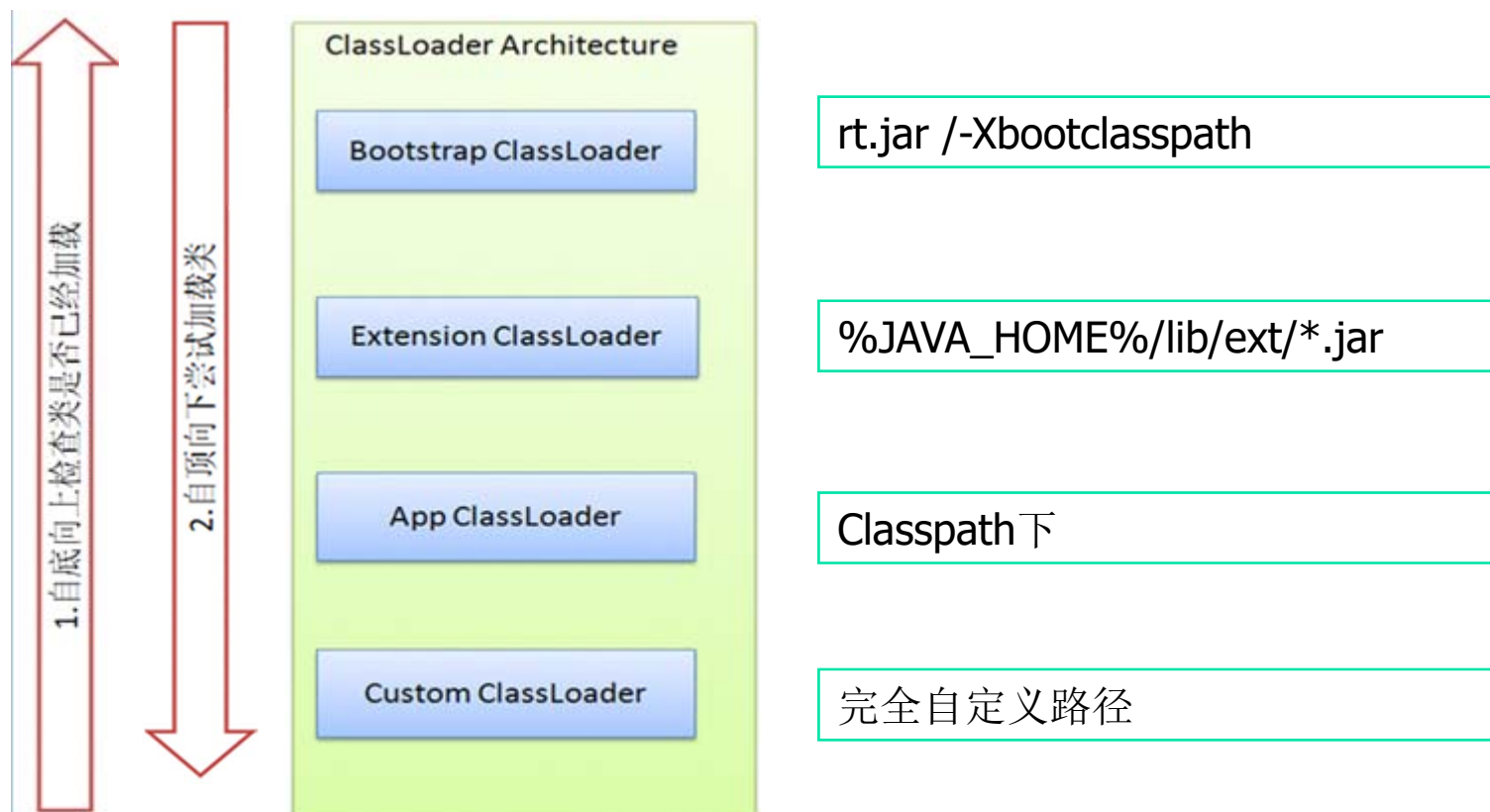
- ① **public Class<?> loadClass(String name) throws ClassNotFoundException**
 - 载入并返回一个Class
- ② **protected final Class<?> defineClass(byte[] b, int off, int len)**
 - 定义一个类，不公开调用
- ③ **protected Class<?> findClass(String name) throws ClassNotFoundException**
 - **loadClass**回调该方法，自定义**ClassLoader**的推荐做法
- ④ **protected final Class<?> findLoadedClass(String name)**
 - 寻找已经加载的类



3.1 class装载验证流程

1. **BootStrap ClassLoader**（启动**ClassLoader**）
2. **Extension ClassLoader**（扩展**ClassLoader**）
3. **App ClassLoader**（应用**ClassLoader**/系统**ClassLoader**）
4. **Custom ClassLoader**(自定义**ClassLoader**)
5. 每个**ClassLoader**都有一个**Parent**作为父亲

3.1 class 装载验证流程





3.1 class装载验证流程

```
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been loaded
        Class c = findLoadedClass(name);
        if (c == null) {
            long t0 = System.nanoTime();
            try {
                if (parent != null) {
                    c = parent.loadClass(name, false);
                } else {
                    c = findBootstrapClassOrNull(name);
                }
            }
            catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }
        }
    }
}
```

3.1 class 装载验证流程

geym.jvm.ch6.findorder
FindClassOrder.java
HelloLoader.java

```
public class HelloLoader {  
    public void print(){  
        System.out.println("I am in apploader");  
    }  
}
```

```
public class HelloLoader {  
    public void print(){  
        System.out.println("I am in bootloader");  
    }  
}
```

```
public class FindClassOrder {  
    public static void main(String args[]){  
        HelloLoader loader=new HelloLoader();  
        loader.print();  
    }  
}
```

D:\tmp\clz\geym\jvm\ch6\findorder
名称
HelloLoader.class
helloloader.txt



3.1 class装载验证流程

1. 直接运行以上代码:
 - ① **I am in apploader**
2. 加上参数 **-Xbootclasspath/a:D:/tmp/clz**
 - ① **I am in bootloader**
 - ② 此时**AppLoader**中不会加载**HelloLoader**
 - **I am in apploader** 在**classpath**中却没有加载
 - 说明类加载是从上往下的

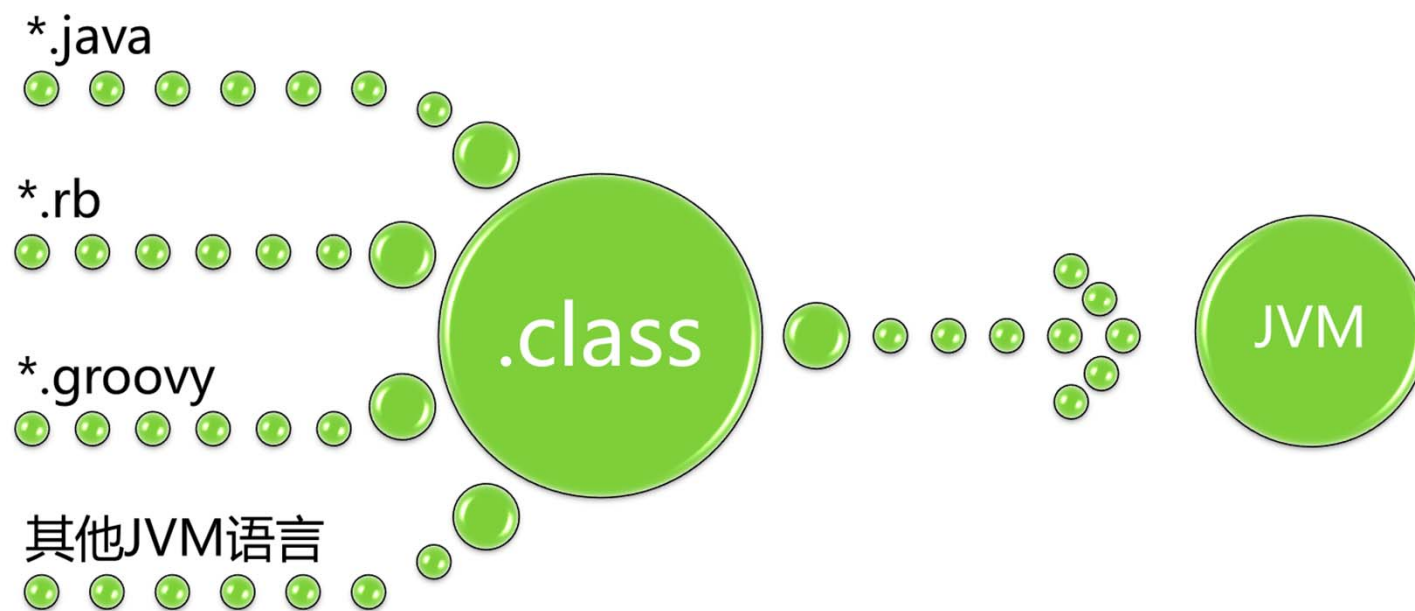


3.1 class文件结构

1. 语言无关性
2. 文件结构
 - ① 魔数
 - ② 版本
 - ③ 常量池
 - ④ 访问符
 - ⑤ 类、超类、接口
 - ⑥ 字段
 - ⑦ 方法
 - ⑧ 属性

3.1 class文件结构

语言无关性:





3.1 class文件结构

类型	名称	数量
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count - 1
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attribute_count	1
attribute_info	attributes	attributes_count

3.1 class文件结构

1. **magic u4**
0xCAFEBAFE
2. **minor_version u2**
3. **major_version u2**

JDK 编译器版本	target 参数	十六进制 minor.major	十进制 major.minor
jdk1.1.8	不能带 target 参数	00 03 00 2D	45.3
jdk1.2.2	不带(默认为 -target 1.1)	00 03 00 2D	45.3
jdk1.2.2	-target 1.2	00 00 00 2E	46.0
jdk1.3.1_19	不带(默认为 -target 1.1)	00 03 00 2D	45.3
jdk1.3.1_19	-target 1.3	00 00 00 2F	47.0
j2sdk1.4.2_10	不带(默认为 -target 1.2)	00 00 00 2E	46.0
j2sdk1.4.2_10	-target 1.4	00 00 00 30	48.0
jdk1.5.0_11	不带(默认为 -target 1.5)	00 00 00 31	49.0
jdk1.5.0_11	-target 1.4 -source 1.4	00 00 00 30	48.0
jdk1.6.0_01	不带(默认为 -target 1.6)	00 00 00 32	50.0
jdk1.6.0_01	-target 1.5	00 00 00 31	49.0
jdk1.6.0_01	-target 1.4 -source 1.4	00 00 00 30	48.0
jdk1.7.0	不带(默认为 -target 1.6)	00 00 00 32	50.0
jdk1.7.0	-target 1.7	00 00 00 33	51.0
jdk1.7.0	-target 1.4 -source 1.4	00 00 00 30	48.0



3.1 class文件结构

1. constant_pool_count u2

2. constant_pool cp_info

①	CONSTANT_Utf8	1	UTF-8编码的Unicode字符串
②	CONSTANT_Integer	3	int类型的字面值
③	CONSTANT_Float	4	float类型的字面值
④	CONSTANT_Long	5	long类型的字面值
⑤	CONSTANT_Double	6	double类型的字面值
⑥	CONSTANT_Class	7	对一个类或接口的符号引用
⑦	CONSTANT_String	8	String类型字面值的引用
⑧	CONSTANT_Fieldref	9	对一个字段的符号引用
⑨	CONSTANT_Methodref	10	对一个类中方法的符号引用
⑩	CONSTANT_InterfaceMethodref	11	对一个接口中方法的符号引用
⑪	CONSTANT_NameAndType	12	对一个字段或部分符号引用



3.1 class文件结构-常量池

CONSTANT_Utf8

- ① tag 1
- ② length u2
- ③ bytes[length]

```
Length of byte array: 3  
Length of string: 3  
String: ()I
```

```
Length of byte array: 20  
Length of string: 20  
String: ()Ljava/lang/Object;
```

```
Length of byte array: 9  
Length of string: 9  
String: compareTo
```



3.1 class文件结构-常量池

CONSTANT_Integer

- ① tag 3
- ② byte u4

```
public static final int sid=99;
```

```
Bytes:    0x00000063  
Integer:  99
```

3.1 class文件结构-常量池

CONSTANT_String

- ① tag 8
- ② string_index u2 (指向utf8的索引)

```
public static final String sname="geym";
```

String: cp info #16 <geym>

The diagram illustrates the mapping from a String constant pool entry to its UTF-8 encoding in the class file constant pool. On the left, a String entry is shown with the tag 'cp info #16' and the value '<geym>'. A green arrow points from this entry to the constant pool entry '[16] CONSTANT_Utf8_info' in the right panel. The right panel shows a list of constant pool entries, with '[16] CONSTANT_Utf8_info' highlighted. To the right of the list, the details for this entry are shown: 'Length of byte array: 4', 'Length of string: 4', and 'String: geym'.

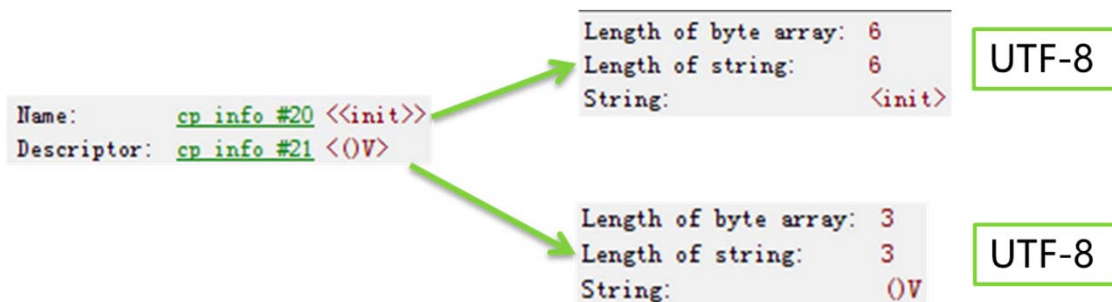
Index	Tag	Value
[09]	CONSTANT_Utf8_info	
[10]	CONSTANT_Utf8_info	
[11]	CONSTANT_Utf8_info	
[12]	CONSTANT_Integer_info	
[13]	CONSTANT_Utf8_info	
[14]	CONSTANT_Utf8_info	
[15]	CONSTANT_String_info	
[16]	CONSTANT_Utf8_info	

Length of byte array: 4
Length of string: 4
String: geym

3.1 class文件结构-常量池

CONSTANT_NameAndType

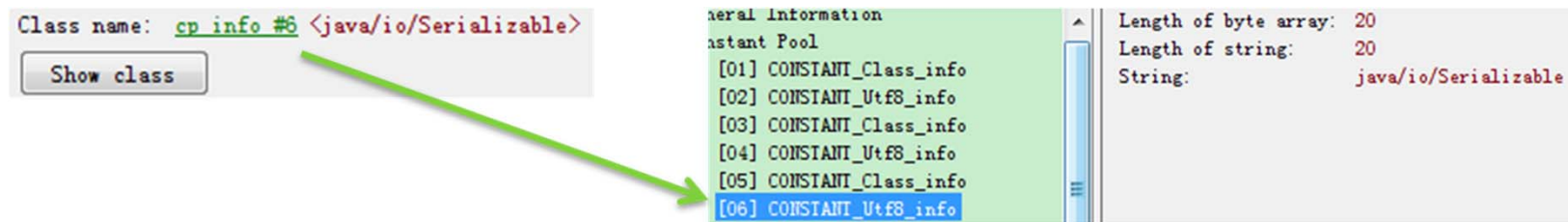
- ① tag 12
- ② name_index u2 (名字, 指向utf8)
- ③ descriptor_index u2 (描述符类型, 指向utf8)



3.1 class文件结构-常量池

CONSTANT_Class

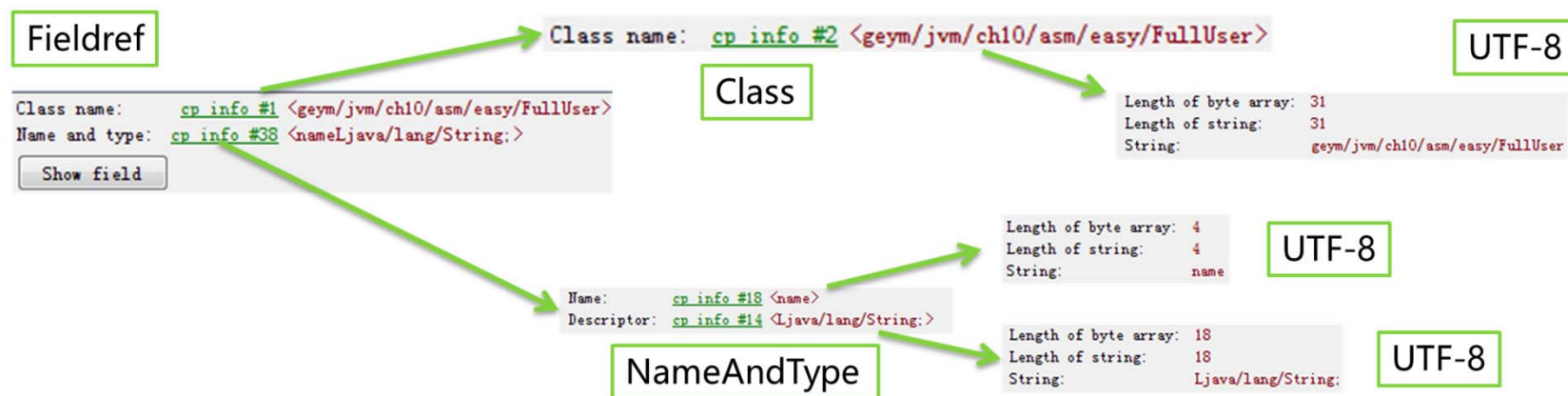
- ① tag 7
- ② name_index u2 (名字, 指向utf8)



3.1 class文件结构-常量池

CONSTANT_Fieldref ,CONSTANT_Methodref ,CONSTANT_InterfaceMethodref

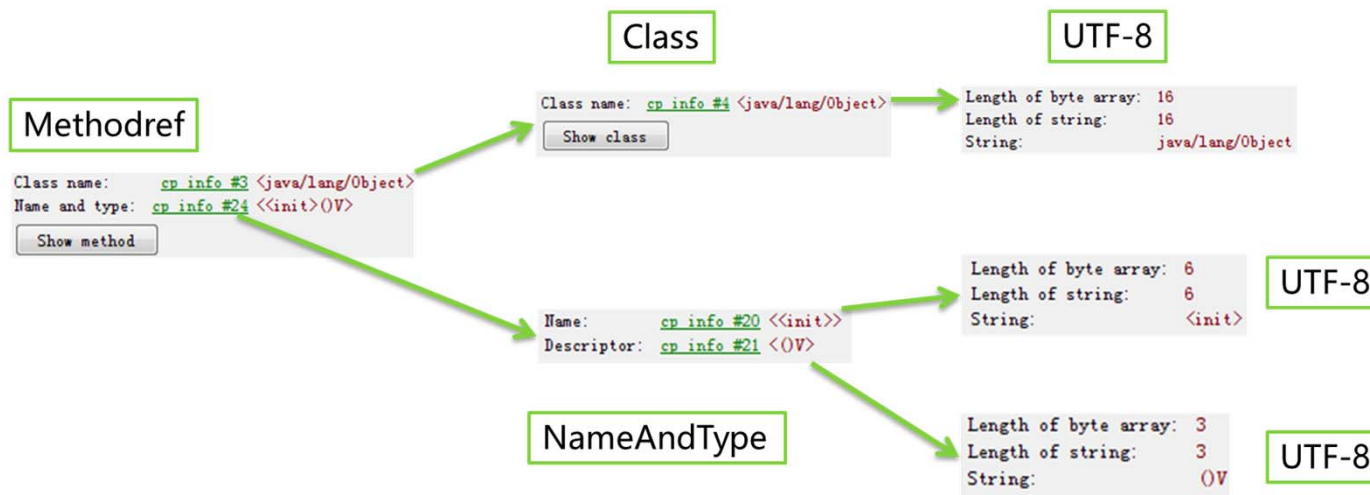
- ① tag 9, 10, 11
- ② class_index u2 (指向CONSTANT_Class)
- ③ name_and_type_index u2 (指向CONSTANT_NameAndType)



3.1 class文件结构-常量池

CONSTANT_Fieldref ,CONSTANT_Methodref ,CONSTANT_InterfaceMethodref

- ① **tag 9 ,10, 11**
- ② **class_index u2 (指向CONSTANT_Class)**
- ③ **name_and_type_index u2 (指向CONSTANT_NameAndType)**



3.1 class文件结构

Offset	小版本							大版本							常量池class name				utf8	string len=27	
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	个数36	info	index		
00000000	CA	FE	BA	BE	00	00	00	33	00	25	07	00	02	01	00	1B					Ëp%...3.%.....
00000010	67	65	79	6D	2F	6A	76	6D	2F	63	68	31	30	2F	61	73					geym/jvm/ch10/as
00000020	6D	2F	65	61	73	79	2F	55	73	65	72	07	00	04	01	00					m/easy/User.....
00000030	10	6A	61	76	61	2F	6C	61	6E	67	2F	4F	62	6A	65	63					.java/lang/Objec
00000040	74	01	00	02	69	64	01	00	01	49	01	00	04	6E	61	6D					t...id...I...nam
00000050	65	01	00	12	4C	6A	61	76	61	2F	6C	61	6E	67	2F	53					e...Ljava/lang/S
00000060	74	72	69	6E	67	3B	01	00	03	61	67	65	01	00	06	3C					tring;...age...<
00000070	69	6E	69	74	3E	01	00	03	28	29	56	01	00	04	43	6F					init>...()V...Co
00000080	64	65	0A	00	03	00	0E	0C	00	0A	00	0B	01	00	0F	4C					de.....L
00000090	69	6E	65	4E	75	6D	62	65	72	54	61	62	6C	65	01	00					ineNumberTable..
000000A0	12	4C	6F	63	61	6C	56	61	72	69	61	62	6C	65	54	61					.LocalVariableTa
000000B0	62	6C	65	01	00	04	74	68	69	73	01	00	1D	4C	67	65					ble...this...Lge
000000C0	79	6D	2F	6A	76	6D	2F	63	68	31	30	2F	61	73	6D	2F					ym/jvm/ch10/asm/
000000D0	65	61	73	79	2F	55	73	65	72	3B	01	00	05	67	65	74					easy/User;...get
000000E0	49	64	01	00	03	28	29	49	09	00	01	00	16	0C	00	05					Id...()I.....
000000F0	00	06	01	00	05	73	65	74	49	64	01	00	04	28	49	29				setId...(I)
00000100	56	01	00	07	67	65	74	4E	61	6D	65	01	00	14	28	29					V...getName...()
00000110	4C	6A	61	76	61	2F	6C	61	6E	67	2F	53	74	72	69	6E					Ljava/lang/Strin
00000120	67	3B	09	00	01	00	1C	0C	00	07	00	08	01	00	07	73					g;.....s
00000130	65	74	4E	61	6D	65	01	00	15	28	4C	6A	61	76	61	2F					etName...(Ljava/
00000140	6C	61	6E	67	2F	53	74	72	69	6E	67	3B	29	56	01	00					lang/String;)V..
00000150	06	67	65	74	41	67	65	09	00	01	00	21	0C	00	09	00					.getAge....!....
00000160	06	01	00	06	73	65	74	41	67	65	01	00	0A	53	6F	75				setAge...Sou
00000170	72	63	65	46	69	6C	65	01	00	09	55	73	65	72	2E	6A					rceFile...User.i



3.1 JVM字节码执行过程

javap

class文件反汇编工具

```
public class Calc {  
    public int calc() {  
        int a = 500;  
        int b = 200;  
        int c = 50;  
        return (a + b) / c;  
    }  
}
```

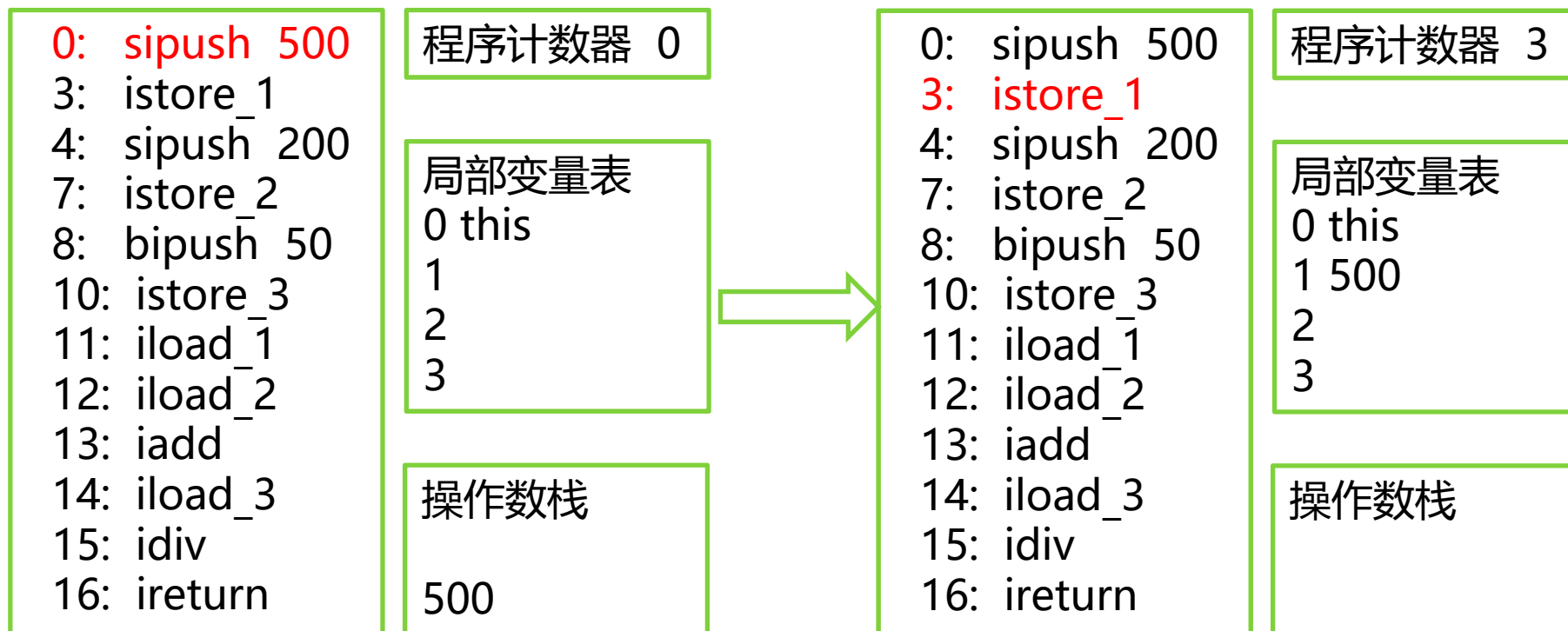
javap -verbose Calc



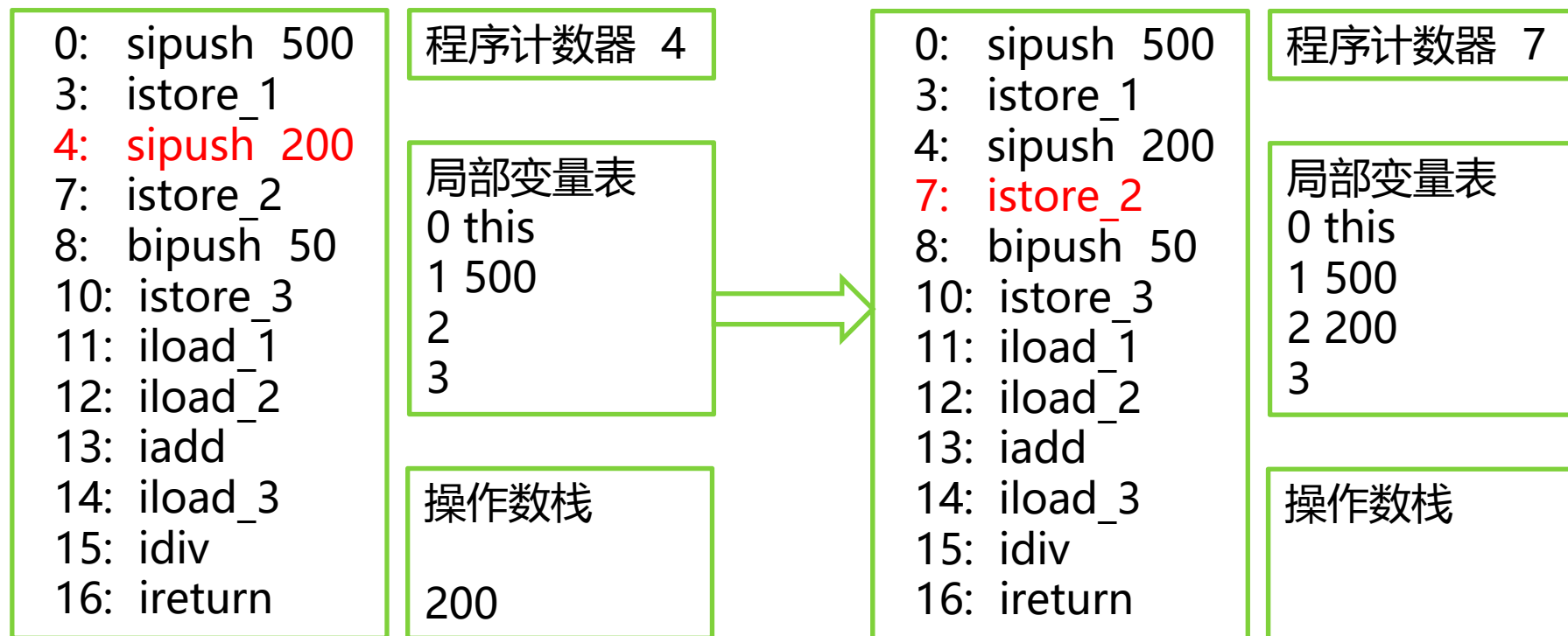
3.1 JVM字节码执行过程

```
public int calc();  
Code:  
Stack=2, Locals=4, Args_size=1  
0: sipush 500  
3: istore_1  
4: sipush 200  
7: istore_2  
8: bipush 50  
10: istore_3  
11: iload_1  
12: iload_2  
13: iadd  
14: iload_3  
15: idiv  
16: ireturn  
}
```

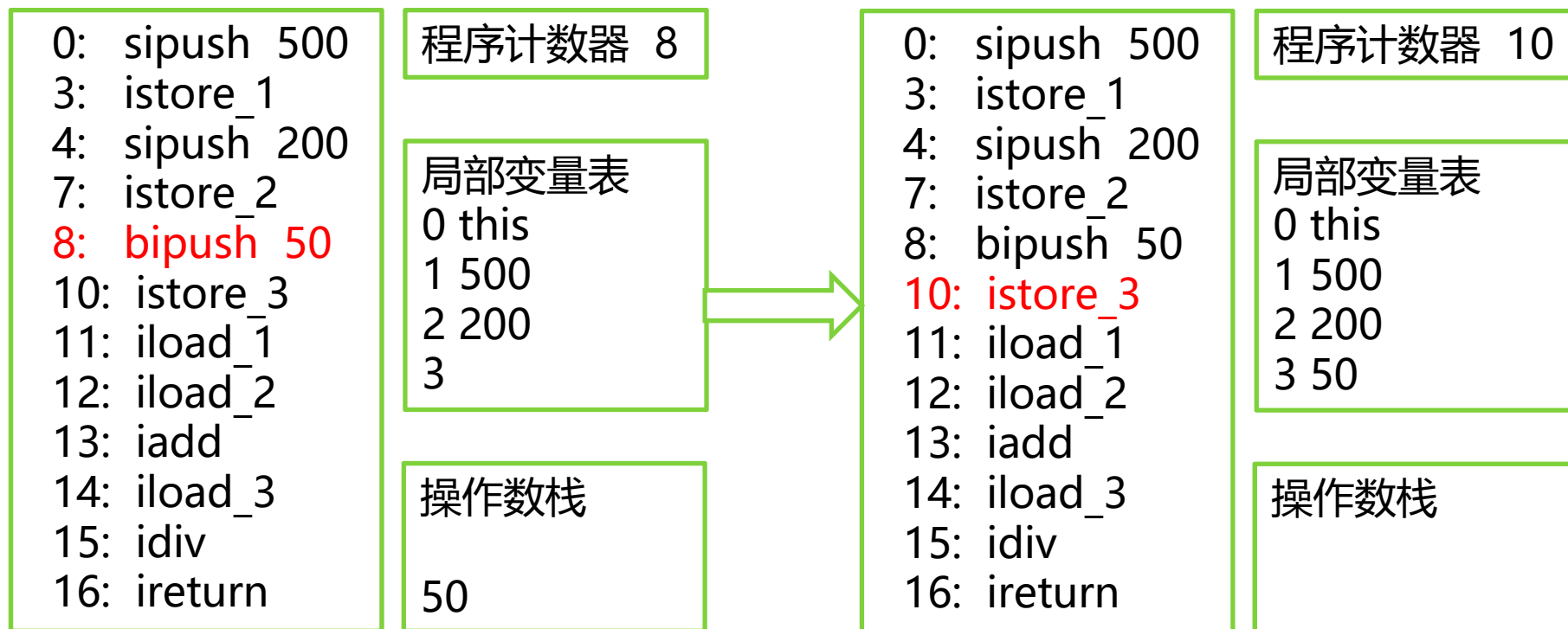
3.1 JVM字节码执行过程



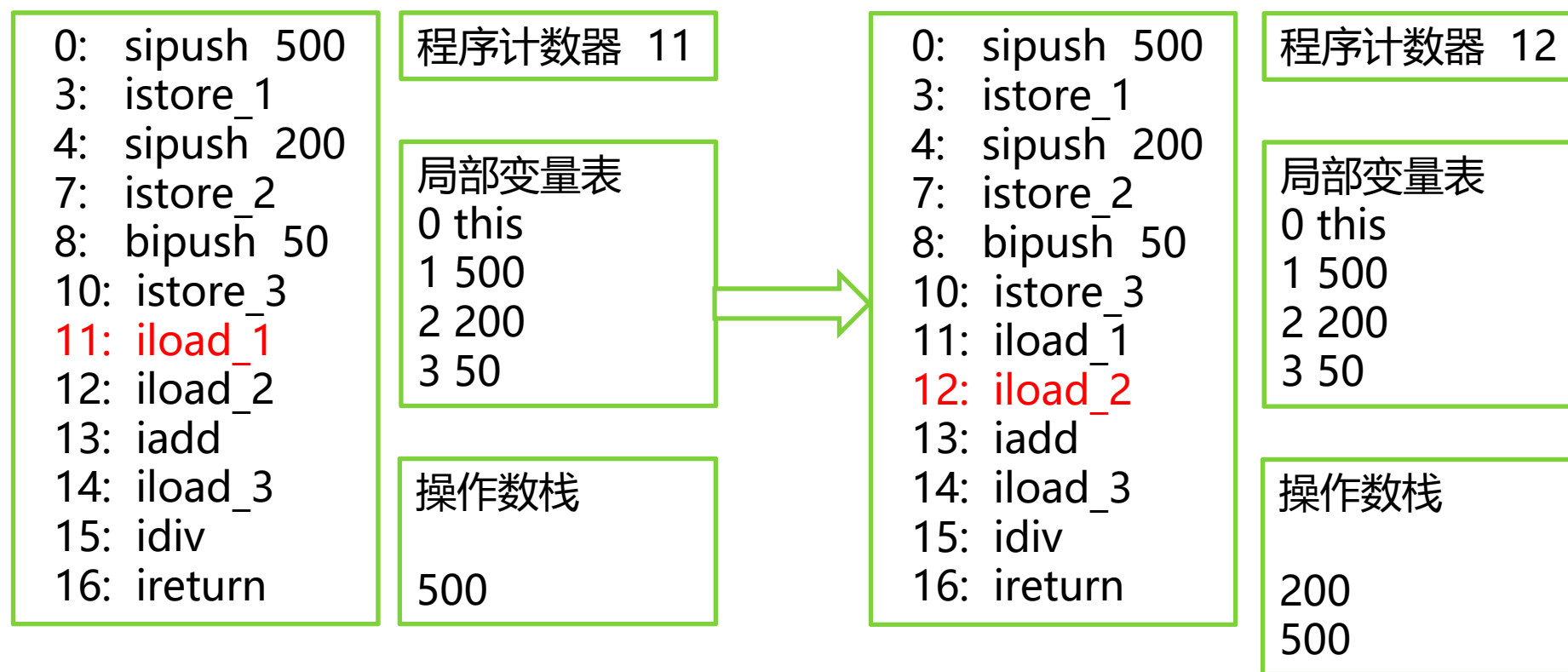
3.1 JVM字节码执行过程



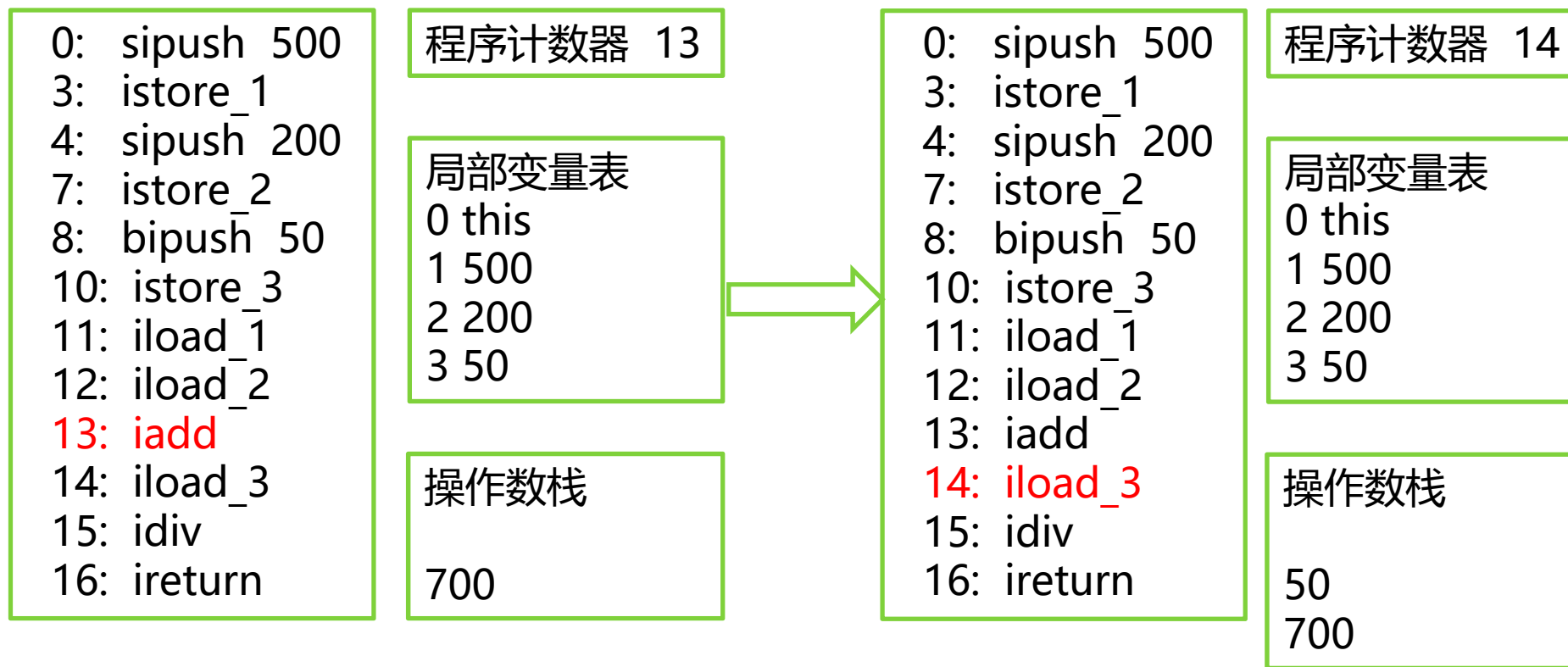
3.1 JVM字节码执行过程



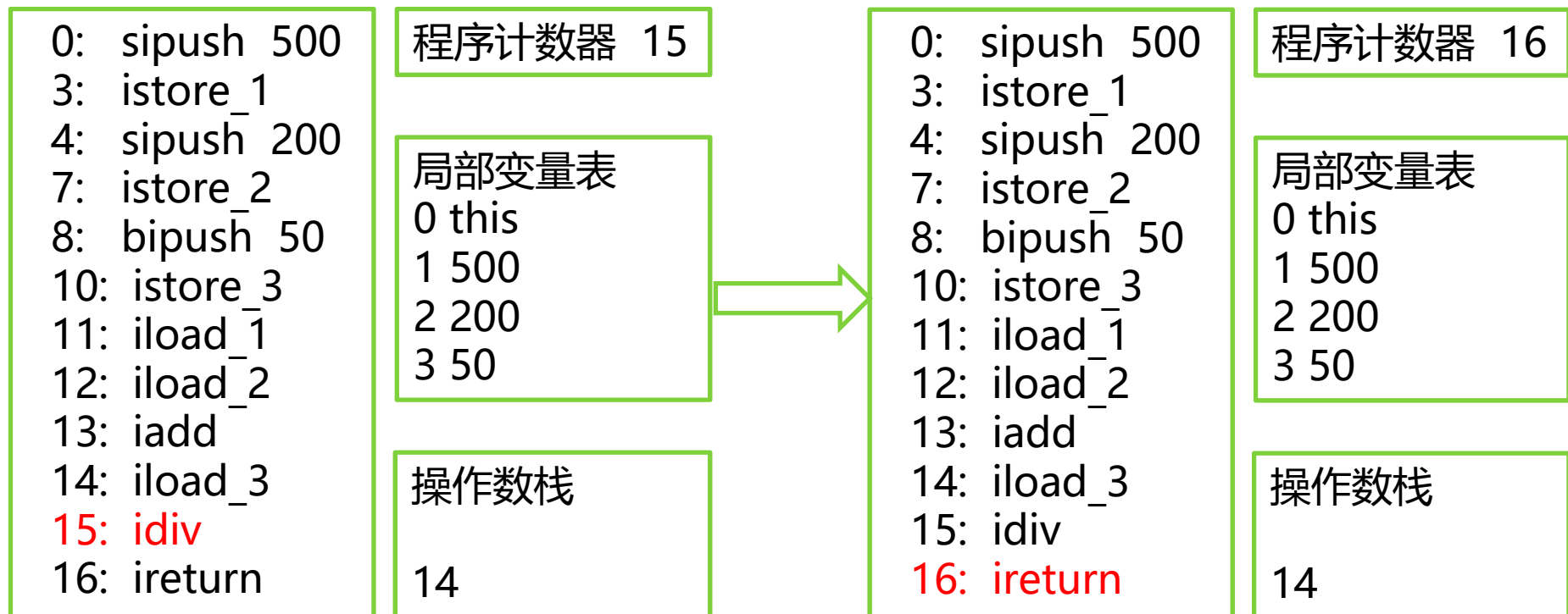
3.1 JVM字节码执行过程



3.1 JVM字节码执行过程



3.1 JVM字节码执行过程



3.1 Java程序内存分析

- 参见“代码内存分析.pdf”中的内容
- **TestClassandObject.java**
- **TestMemory1.java**

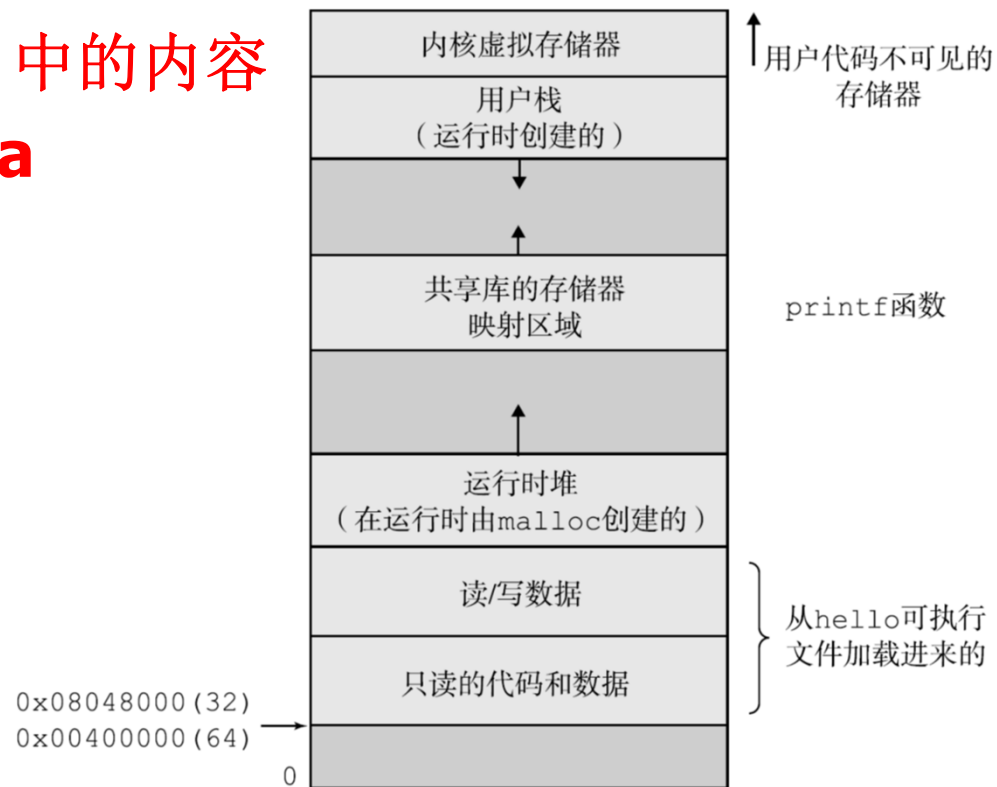


图 1-13 进程的虚拟地址空间



3.2 类的封装性

1. 3.2.1 构造方法与析构方法
2. 3.2.2 对象的引用和运算
3. 3.2.3 访问控制-隐藏/封装
4. 3.2.4 静态成员



3.2.1 构造方法

构造方法，又称为构造器**Constructor**，用于构造该类的实例。

- 格式：[修饰符] 类名 (形参列表) {...}
- 通过**new**关键字调用；
- 构造器虽然有返回值，但是不能定义返回类型（返回值的类型肯定是本类），不能在构造器里调用**return**；
- 如果我们没有定义构造器，系统会自动定义一个无参的构造器，如果已定义则编译器不会添加；
- 构造器的方法名必须和类名一致。
- 作用：构造该类的对象，经常也用来初始化对象的属性。



3.2.1 构造方法与析构方法

1. 声明及调用构造方法

```
public class MyDate
{   public MyDate(int y, int m, int d)
    {
        year = y;
        month = m;
        day = d;
    }
}

MyDate d = new MyDate(2009,7,18);
```

2. 默认构造方法

```
public MyDate()
```



构造方法重载

方法的重载overload。方法的重载是指一个类中可以定义有相同的名字，但参数不同的多个方法。调用时，会根据不同的参数列表选择对应的方法。

- 两同三不同：同一个类，同一个方法名，不同的参数列表（类型、个数、顺序不同）。
- 只有返回值不同不构成方法的重载。
- 只有形参的名称不同，不构成方法的重载。
- 构造方法与普通方法一样也可以重载。



构造方法重载

重载的含义，**Math.abs()**方法声明有**4**种：

- ① **int abs(int a)**
- ② **long abs(long a)**
- ③ **float abs(float a)**
- ④ **double abs(double a)**

void print(boolean b)

void print(char c)

void print(int i)

void print(double d)



3.2.2 对象的引用和运算

1. **this**引用，成员方法的一个隐式参数。

① 指代对象本身

- **this**

② 访问本类的成员变量和成员方法

- **this**.成员变量

- **this**.成员方法([参数列表])

③ 调用本类重载的构造方法

- **this**([参数列表])

3. **instanceof**对象运算符

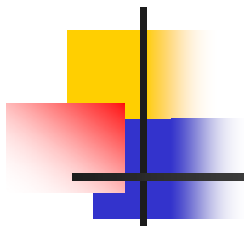
- **d instanceof MyDate**



3.2.2 对象的引用和运算

this（成员方法的一个隐式参数，还有一个隐式参数叫**super**（父类对象的引用））关键字。调用非静态方法时，自动传递了这两个隐式参数！！必须传这两个隐式参数，因为类的所有的实例化对象都共享一份类代码（包括方法代码），如果不传，则在方法中不知道是哪个对象调用的！

- 普通方法中，**this**总是指向调用该方法的对象；
- 构造方法中，**this**总是指向正要初始化的对象；
- **this**不能用于**static**方法；
- 可以在一个构造方法中通过**this**调用其它构造方法，且必须是构造方法中的第一条语句。



构造方法是创建**Java**对象的重要途径，通过**new**关键字调用构造器时，构造器也确实返回该类的对象，但是这个对象并不完全由构造器负责创建。实际上，创建一个对象分为如下四步：

- 分配对象空间，并将对象成员变量初始化为**0**或空；
- 执行属性值的显式初始化；
- 执行构造方法；
- 返回对象的地址给相关的变量。

this只能用在方法中，其的本质是“调用该方法的创建好的对象的地址”。由于在构造方法调用时，对象已经创建。因此，在构造方法中也可以使用**this**代表“当前对象”。



3.2.3 隐藏/封装

为什么需要封装？封装的作用和含义？

- 隐藏对象内部的复杂性，只对外公开简单的接口。便于外界调用，从而提高系统的可扩展性、可维护性。
- 我们程序设计要追求“高内聚、低耦合”（只提供最简单的接口给别人，内部复杂的细节不给别人看，也不允许别人干预）。高内聚就是内的内部数据操作细节自己完成，不允许外部干涉；低耦合就是仅暴露少量的方法给外部使用。

3.2.3 隐藏/封装

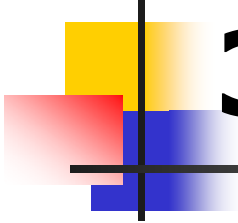
类的访问权限：只有**public**和缺省两种。

```
public class MyDate  
class MyDate_ex
```

2. 类中成员的访问权限

权限修饰符	同一类	同一包	不同包的子类	所有类
public (公有)	✓	✓	✓	✓
protected (受保护的)	✓	✓	✓	
缺省	✓	✓		
private (私有)	✓			

3. 声明**set()**和**get()**方法存取对象的属性



3.2.3 隐藏/封装

封装要点：

- **类的属性的处理**：一般使用**private**（除非本属性确定会让子类继承）；提供相应的**get/set**方法来访问相关属性，这些方法通常是**public**，从而提供对属性的读取操作。（**boolean**变量的**get**方法是**is**开头）；常量或**static**变量公开。
- 一些只用于本类的辅助性方法可以用**private**，希望其它类调用的方法可以用**public**。
- **default**：默认访问控制属性，什么都没加就是该控制符。有的书上说**friendly**、**package**，这都是一个意思，都不能真的写出来，如果什么访问修饰符都不加，就是**default/friendly/package**。
- **Java**的访问控制是**停留在编译层**，也就是它不会在**.class**文件中留下任何痕迹，只在编译的时候进行访问控制的检查。其实，通过反射的手段，可以访问任何包下任何类中的成员，例如，访问类中的私有成员也是可以的。



3.2.4 静态成员

静态成员变量。在类中，用**static**声明的成员变量为静态成员变量，或叫做类属性、类变量。它为该类的公用变量，属于类，被该类的所有实例共享（只此一份），在类被载入时被显式初始化。可以使用“对象.类属性”来调用，不过一般都使用“类名.类属性”。**static**变量置于方法区。

静态方法。用**static**声明的方法为静态方法。不需要对象，就可以调用（“类名.方法名”）；在调用该方法时，不会将**对象的引用（this）**传递给它，**所以在static方法中不可访问非static的成员。**

静态初始化块 static {}：如果希望加载后，对整个类进行某些初始化操作，可以使用**static**初始化块

- 是在类初始化时执行，不是在创建对象时执行。
- 静态初始化块中不能访问非**static**成员变量。
- 执行顺序：上溯到**Object**类，先执行**Object**的静态初始化块，再向下执行子类的静态初始化块，直到我们的类的静态初始化块为止。



3.2.4 静态成员

static的一些要点:

- **static**修饰的方法，**不能直接访问**本类中的非静态（**static**）成员（包括方法和属性），本类的非静态方法可以访问本类的静态成员（包括方法和属性）静态方法要慎重使用（原因是静态变量会一直存在，占用资源），且在静态方法中不能出现**this**关键字。
- 父类中是静态方法，子类中不能覆盖为非静态方法；在符合覆盖规则的前提下，在父子类中，父类中的静态方法可以被子类中的静态方法覆盖，但无多态。（在使用对象调用静态方法时，实则是调用编译时类型的静态方法）。
- 父子类中，静态方法只能被静态方法覆盖，父子类中，非静态方法只能被非静态方法覆盖。
- 类中的实例变量是在创建对象时被初始化的，被**static**修饰的属性（类变量），是在类加载时被创建并进行初始化，类加载的过程只进行一次。也就是类变量只会被创建一次。



3.3 类的继承性

1. 3.3.1 由继承派生类
2. 3.3.2 继承原则及作用
3. 3.3.3 子类的构造方法



3.3 类的继承性

(1) 从面向对象设计**OOD**的角度来说，类是对对象的抽象、继承是对某一批类的抽象，从而实现对现实世界更好的建模；从面向对象编程**OOP**的角度来说，这种组织方式可以提高代码的复用性。

(2) **extends**的意思是“扩展”。子类是父类的扩展。

- **human**类（人类）继承**primate**类（灵长类）、**primate**类继承**mammal**类（哺乳类）、哺乳类继承**animal**类（动物类）；
- **monkey**（猴子类）也继承**primate**类。人类和猴子类不仅继承灵长类，也继承哺乳类、动物类；
- **reptile**（爬行类）和**insect**（昆虫类）也继承**animal**类。鳄鱼类（**corcodile**）和蛇类（**snake**）也继承爬行类。蝴蝶类（**butterfly**）也继承昆虫类；
- 动物：**digest food**（消化食物）、**through the blastula stage**（胚胎阶段）；
- 哺乳动物：**warm blooded**（恒温）、**mammary glands**（乳腺）；
- 灵长动物：**five fingers**（5根手指）、**opposable thumbs**（一对手指）、**fingernails**（指甲）；
- 人类：**highly developed brains**（发达的大脑）、**erect body carriage**（直立姿态）。



3.3 类的继承性

(3) 子类继承父类，可以得到父类的全部属性和方法（除了父类的构造方法）。但不一定可以直接访问（例如父类中声明为私有的属性和方法）。**java.lang.Object**类是所有类的根类。

(4) 在**java (C++-)**中类只有单继承，没有像**C++**那样的多继承。多继承，就是为了实现代码的复用性，却引入了复杂性，使得系统类之间的关系混乱；**Java**中的多继承，可以通过接口实现。



3.3 类的继承性

(5) 方法的重写 (**override**) .

- 在子类中可以根据需要对父类中继承来的方法进行重写。
- 重写方法必须和被重写方法具有相同方法名称、参数列表 (形成重写的要点1)。通过子类去调用该方法, 会调用重写方法而不是被重写方法 (叫做重写方法覆盖被重写方法)。
- 重写方法的访问权限, 子类大于等于父类 (由于多态) (形成重写的要点2)。
- 重写方法的返回值类型和声明异常类型, 子类小于等于父类 (形成重写的要点3)。
- 可以在子类重写方法中调用被重写方法: **super**关键字。
- 与重载 (**overload**) 没有任何关系。方法重载指的是: 同一个类中, 一个方法名对应了多个方法(形参列表不同); 方法的重写指的是: 子类重写了父类的方法!
- 利用重写, 既可以重用父类的方法, 而且还可以灵活的扩充。
- 对象.方法(): 先在本类内部找是否有该方法, 如果没有, 到直接父类去找, 如果还没有, 则一直往上层找, 一直找到**Object**, 如果还没有, 则报错。



3.3.1 由继承派生类

[修饰符] **class** 类 [**extends** 父类]
[**implements** 接口列表]

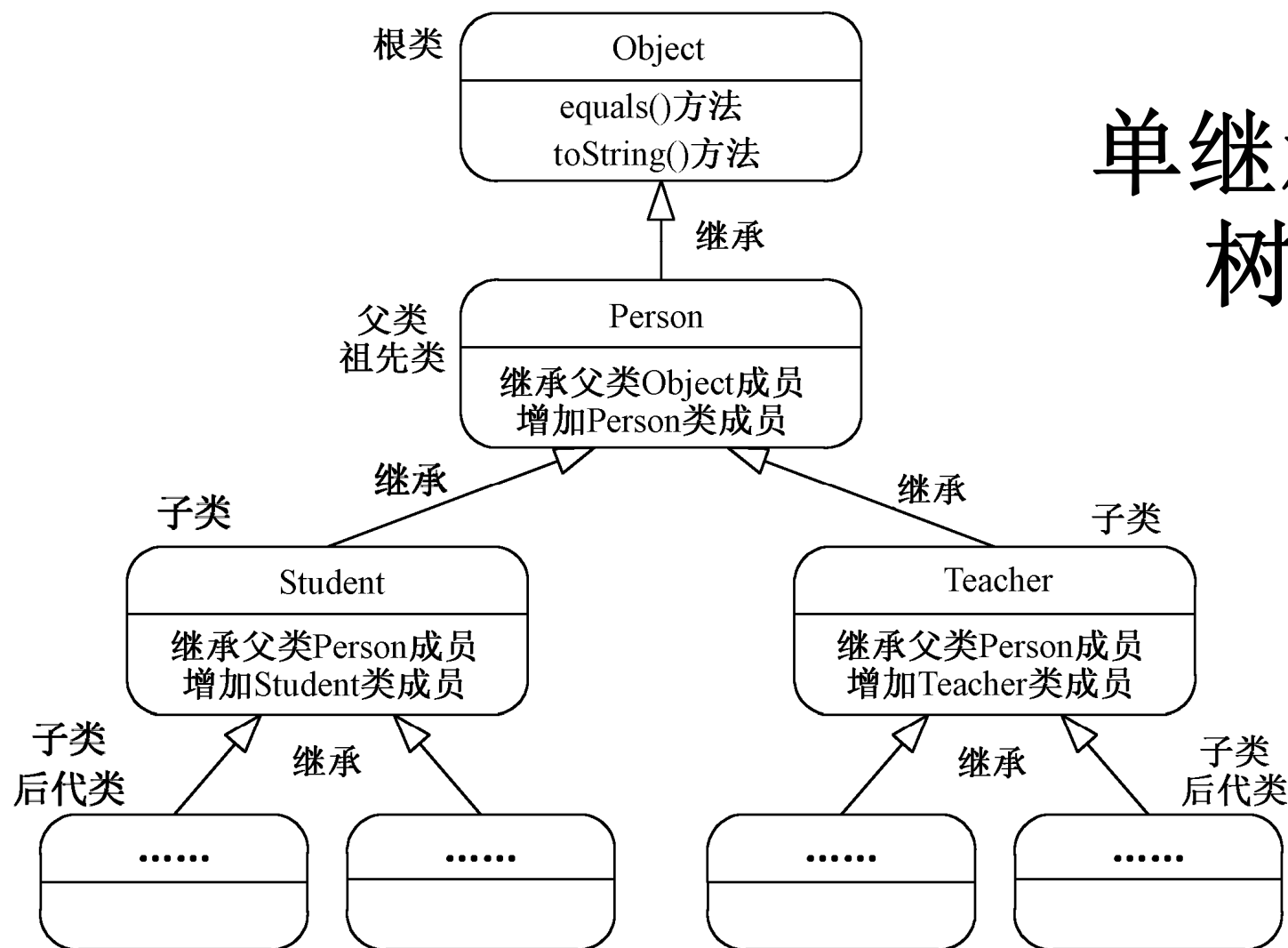
```
public class Student extends Person
{
    String speciality;           //专业
}
```

3.3.2 继承原则及作用

1. 继承原则

- ① 子类继承父类的成员变量
- ② 子类继承父类除构造方法以外的成员方法
- ③ 子类不能继承父类的构造方法
- ④ 子类可以增加成员，可以重定义从父类继承来的成员，但不能删除它们。

2. 继承的作用



单继承，
树结构



3. Object类

```
public class Object
{
    public Object()                //构造方法
    public String toString()       //描述对象
    public boolean equals(Object obj) //比较对象相等
    protected void finalize() throws Throwable
}
```

Java中的类都是**Object**的子类

```
public class Person extends Object
```

如果定义一个类时，没有调用**extends**，则它的父类是**java.lang.Object**。**Object**类是所有**java**类的根基类。



4. 子类对父类成员的访问权限

- ① 子类不能访问父类的私有成员（**private**）。
- ② 子类能够访问父类的公有成员（**public**）和保护成员（**protected**）。
- ③ 子类对父类的缺省权限成员的访问控制，以包为界分两种情况，可以访问当前包中



3.3.3 子类的构造方法

1. 使用**super()**调用父类构造方法

super([参数列表])

```
public Student(String name, MyDate birthday, String spec)
```

```
{
```

```
    super(name, birthday);
```

//调用父类同参数的构造方法

```
    this.speciality = spec;
```

```
}
```

2. 默认执行**super()**

```
public Student()
```

//Java提供的默认构造方法

```
{
```

```
    super();
```

//调用父类构造方法**Person()**

```
}
```



子类的构造方法没有调用**super()**或**this()**，将默认执行**super()**

```
public Person()  
{ super();           // 调用父类构造方法Object()  
}  
  
public Student()  
{  
    super();           // 默认调用  
    speciality="";  
}
```



super引用

1. 调用父类的构造方法

super([参数列表])

2. 引用父类同名成员

① 子类隐藏父类成员变量

- **super.**成员变量

② 子类覆盖父类成员方法

- **super.**成员方法([参数列表])



super总结

- **super**是直接父类对象的引用。可以通过**super**来访问父类中被子类覆盖的方法或属性。
- **super**和**this**一样，只能用于方法内部，是方法的两个隐式参数。
- 普通方法：没有顺序限制，可以随便调用。
- 构造方法：任何类的构造方法中，若是构造函数的第一行代码没有显式调用**super(...)**；那么**Java**默认都会调用**super()**；作为父类的初始化函数。所以这里的**super()**加不加都会无所谓。（内存分析，**wrap:new**对象的时候采用子类包裹父类的结构）
- 同一个构造方法里面不能同时调用**super()**和**this()**。
- 在本类构造方法中通过**super()**调用，会一直上溯到**Object()**这个构造函数，然后按类层级，依次向下执行各层级构造函数中剩下的代码，直至最低层级的构造函数。同**this()**一样，**super()**方法也应该放到构造方法的第一行。
- **new**一个类的对象的时候，通过构造方法的从上至下的依次调用，就依次建立了新的根对象、父类对象和自身对象，其中，**this**指向新建的对象本身，**super**指向新建的直接父类对象本身。



组合vs继承

- “**is-a**”关系使用**继承**：上面的通过在**Audi**类中增加一个**Car**属性虽然也复用了代码，但是不合逻辑不容易理解。
- “**has-a**”关系使用**组合**：计算机类、主板类。可以通过在计算机类中增加主板属性来复用主板类的代码。
- 如果仅仅从代码复用的角度考虑，组合完全可以替代继承。
- 所谓组合，就是把要组合的另一个类作为属性放到类里面。
- 是就用继承、有就用组合。



3.4 类的多态性

1. 3.4.1 子类重定义父类成员
2. 3.4.2 类型的多态
3. 3.4.3 编译时多态和运行时多态性
4. 3.4.4 多态的方法实现



3.4.1 子类重定义父类成员

1. 子类**隐藏**父类成员变量
2. 子类覆盖父类成员方法

覆盖（**override**）是指子类声明并实现父类中的同名方法并且参数列表也完全相同。

3. 子类继承并重载父类成员方法

重载（**overload**）是指同一个类中的多个方法可以同名但参数列表必须不同。



3.4.2 类型的多态

1. 子类对象即是父类对象

```
new Student() instanceof Person //true  
new Person() instanceof Student //false
```

2. 父类对象引用子类实例

```
Person p = new Student(); //赋值相容  
Student s = new Person(); //语法错  
Object obj = new Person(); //赋值相容
```




3.4.3 编译时多态和运行时多态性

1. 编译时多态性

方法重载都是编译时多态。

方法覆盖表现出两种多态性，当对象引用本类实例时，为编译时多态，否则为运行时多态。

```
Person p = new Person(.....);
```

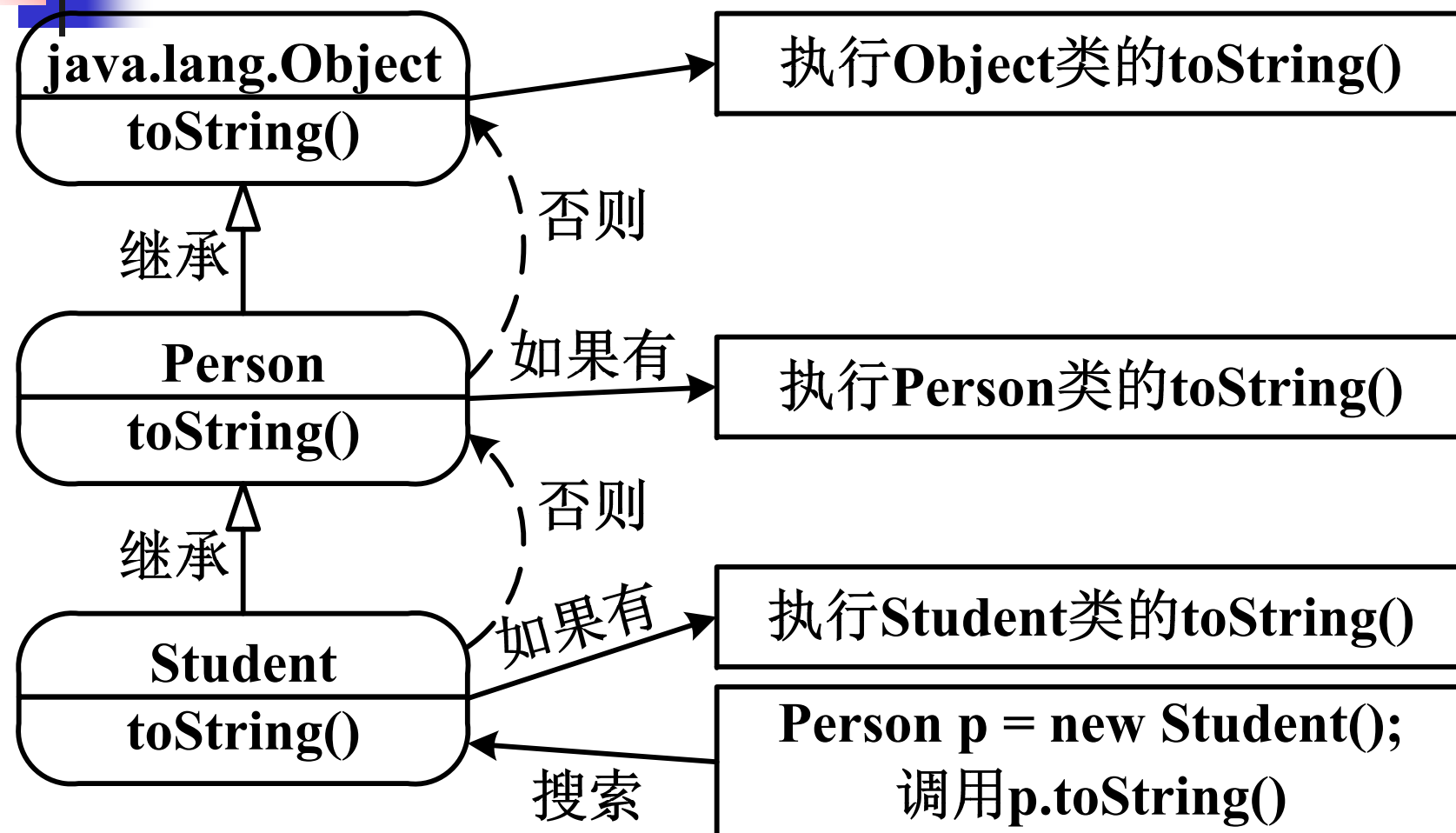
```
p.toString()       //执行Person类的toString()
```

```
Student s= new Student(.....);
```

```
s.toString()       //执行Student类的toString()
```

2. 运行时多态性

图3.9 寻找(new Student()).toString()匹配的执行方法





p不能调用子类增加的方法

```
p.set("经济管理系","信息管理专业","003",true);  
//语法错, Person类没有声明该方法
```



3.4.4 多态的方法实现

1. 多态的**toString()**方法

Object类声明:

```
public String toString() //返回当前对象的信息字符串  
{  
    return this.getClass().getName() + "@"  
        + Integer.toHexString(this.hashCode());  
}
```



2. 多态的**equals(Object)**方法

(1) 子类扩展父类**equals()**方法

设**public boolean equals(Person p)**

① 子类若**public boolean equals(Student s) // 重载**
则**s1.equals(p1) // 继承, 执行父类对象比较规则**

s2.equals(s1) // 重载, 执行子类对象比较规则

结论: 不必要, 因为**Person**参数可接受**Student**实例。

② 子类若**public boolean equals(Person p) // 覆盖**



(2)Object类的equals(Object)方法

```
public boolean equals(Object obj)  
{  
    return this == obj;  
}
```

(3)子类覆盖Object类的 equals(Object)方法

Person类声明**equals(Object)**方法，覆盖，
与父类不可比

```
public boolean equals(Object obj) //比较两个对象的值是否相等
{                                  //覆盖Object类的equals()方法
    if (this==obj)                //this指代调用当前方法的对象
        return true;
    if (obj instanceof Person)    //判断当前对象是否属于Person类
    {
        Person p = (Person)obj;   //强制类型转换
        return this.name.equals(p.name) &&
            this.birthday.equals(p.birthday);
    }
    return false;
}
```



3.4 方法的多态性总结

- **Java**程序经历编译、运行阶段。编译用父类声明，运行时使用具体子类实例化。
- 多态性是**OOP**中的一个重要特性，主要是用来实现动态联编。就是程序的最终状态只有在执行过程中才被决定而非在编译期间就决定。这对于大型系统来说能提高系统的灵活性和扩展性。
- **Java**中如何实现多态：引用变量的两种类型：编译时类型（模糊一点，一般是一个父类），由声明时的类型决定。运行时类型（运行时，具体是哪个子类就是哪个子类），由实际对应的对象类型决定。
- 多态存在的**3**个必要条件：要有继承，要有方法重写，父类引用指向子类对象。
- 多态一般指的是重写方法的多态。
- 继承时，对于方法覆盖时，**new**的谁，**this**就指向谁，多态性决定。如果是成员变量，**this**在哪个类就指向哪个类的成员变量，成员变量没有多态性。



3.4 类的多态性总结

- 方法的多态是子类重写了父类的方法，父类引用指向子类对象，通过父类被重写方法，实际调用的是子类的重写方法（这个相当于把子类对象地址赋给父类对象引用，编译时，编译器通过声明的父类类型去查找父类代码中是否有该方法，有，则通过了编译，实际运行时，则依据实际内存地址找到了子类对象内存空间，因此，实际执行的是子类方法）。
- 上述情况下，如果想通过父类引用来调用子类扩展的方法，则编译时将不通过（因为编译时只认父类引用所声明的父类类型，而父类类型里没有该子类扩展方法），可以将该父类引用强制转型为子类类型来达到目标。
- 引用类型的强制转型，适用于将父类类型向下强制转换为子类类型。不同类型之间不能强制转型（编译不通过）。



3.4 类的多态性总结

- 子类类型的对象地址可以直接赋给父类类型的引用对象，这个称为向上转型，是实现多态的基础。
- **A instanceof B**: **A**对象的类型是否是**B**类型，只有在**A**对象的类型和**B**类型相同，或为父子类型时，编译不报错。而在运行时，只有**A**对象类型为**B**类型的子类型或者就是**B**类型时，结果才返回**true**。
- 内存分析（例子：**myServlet**）：调用父类的**service()**，然后调用子类的**doGet()**（注意：**this**关键字指向整个最终包裹对象，即最外层的子对象；而在包裹对象中，每一层对象通过**super**关键字指向内一层的父对象）。
- 多态指的是方法的多态（到底调用那个方法，运行时决定），属性没有多态。



3.5 类的抽象性

1. 3.5.1 抽象类
2. 3.5.2 最终类



3.5.1 抽象类

(1) 为什么需要抽象类？如何定义抽象类？

- 是一种模版模式。抽象类为所有子类提供了一个通用模版，子类可以在这个模版基础上进行扩展。
- 通过抽象类，可以避免子类设计的随意性。通过抽象类，可以严格限制子类的设计，使子类之间更加通用。

(2) 要点：

- 有抽象方法的类只能定义成抽象类。
- 抽象类不能实例化，即不能用**new**来实例化抽象类。
- 抽象类可以包含属性、方法、构造方法（抽象类除了抽象方法，也可以定义普通方法、构造方法和普通属性）。但是构造方法不能用来**new**实例，只能用来被子类调用。
- 抽象类只能用来继承。
- 抽象方法只有方法的声明，没有方法体（即**{ }**和里面的代码），其必须被子类实现（这也是抽象方法的设计初衷，提供规范，要求各子类实现规范）。
- 抽象类用于设计和实现的分离，但是这种分离不够彻底。
- 一个抽象类可以继承另一个抽象类。



3.5.1 抽象类

1. 声明抽象类与抽象方法

```
public abstract class ClosedFigure
    // 闭合图形抽象类
{
    public abstract double area();
        // 计算面积，抽象方法，以分号";"结束
}
```



3.5.2 抽象类

2. 抽象类的特点

- ① 构造方法、静态成员方法不能被声明为抽象方法。
- ② 一个非抽象类必须实现从父类继承来的所有抽象方法。
- ③ 不能创建抽象类的实例。例如：

```
ClosedFigure g = new ClosedFigure();  
    //语法错
```

3. 抽象类与抽象方法的作用



3.5.1 抽象类

(3) 要点 (续) :

- **abstract**修饰方式的初衷就是要求其子类覆盖（实现）这个方法，并且调用时可以以多态方式调用子类覆盖后的方法（抽象类主要和多态技术相结合），即抽象方法必须在其子类中实现，除非子类本身也是抽象类。**abstract**不允许修饰成员变量，因为成员变量也没有重写这个概念！
- 有抽象方法的类一定是抽象类，但是抽象类中不一定是抽象方法，也可以全部都是具体方法。
- **abstract**修饰的类不能生成对象实例，但可以作为对象变量声明的类型，即编译时类型。抽象类就相当于一类的半成品，需要子类继承并覆盖其中的抽象方法。
- 父类是抽象类，其中有抽象方法，那么子类继承父类，并把父类中的所有抽象方法都实现（覆盖）了，子类才有创建对象的实例的能力，否则子类也必须是抽象类。抽象类中可以有构造方法，是子类在构造子类对象时需要调用的父类（抽象类）的构造方法。
- 不能放在一起的修饰符：**final**和**abstract**，**private**和**abstract**，**static**和**abstract**，因为**abstract**修饰的方法是必须在其子类中实现（覆盖），才能以多态方式调用，以上修饰符在修饰方法时期子类都覆盖不了这个方法，**final**是不可以覆盖，**private**是不能够继承到子类，所以也就不能覆盖，**static**是可以覆盖的，但是在调用时会调用编译时类型的方法，因为调用的是父类的方法，而父类的方法又是抽象的方法，又不能够调用，所以上面的修饰符不能放在一起。



3.5.2 最终类

1. 声明最终类，不能被继承

```
public final class Math           //数学类，最终类  
public class MyMath extends Math //语法错  
public final class Circle extends Ellipse //最终类
```

2. 声明最终方法，不能被子类覆盖

```
public class Circle extends Ellipse //非最终类  
{  
    public final double area()        //最终方法  
}
```




final关键字

- **final**修饰变量时表示常量。变量被**final**修饰，就会变成常量（常量应大写），一旦赋值不能改变（可以在初始化时直接赋值，也可以在构造方法里赋值，只能在这两种方法里二选一，必须为常量赋值）；**final**的常量不会有默认初始值，对于直接在初始化时赋值方式，**final**修饰符常和**static**修饰符一起使用。
- **final**修饰方法（最终方法）时表示该方法不可被子类重写。但是可以被重载。
- **final**修饰类（最终类）时表示修饰的类不能有子类，不能被继承。比如**Math**、**String**。**final**类中的方法也都是**final**的。



3.6 数组

1. 3.6.1 一维数组
2. 3.6.2 二维数组



3.6.1 一维数组

1. 声明一维数组变量

数据类型[] 数组 或 数据类型 数组[]

int a[]; 或 **int[] a;**

2. 使用**new**为数组分配空间

数组 = **new** 数据类型[长度]

a = new int[5];

int a[] = new int[5];

3. 数组长度**length**

数组.**length**



3.6.1 一维数组

4. 数组元素的表示及运算

数组[下标]

a[0], a[1], a[2], a[3], a[4]

a[i] = a[i-2]+a[i-1];//数组元素能够参加运算

5. 数组声明时赋初值

int a[]={1,2,3,4,5};

6. 数组元素的初始化

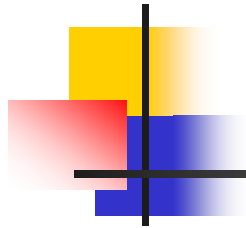
【例2.5】 用一维数组计算**Fibonacci**序列值。



3.6.1 一维数组

注意：

- 数组是相同（不允许出现混合类型）任意数据类型（包括基本类型和引用类型）对象的有序集合。
- 数组也是对象。数组元素相当于对象的成员变量（内存图）。
- 数组长度是确定的，不可变的（一旦被创建，大小不可以被改变），下标从**0**开始。
- 数组是引用类型，它的元素相当于类的实例变量，因此数组一经分配空间，其中每个元素也被按照实例变量同样的方式被隐式初始化（默认初始化）(**0/false/null**)。定义并用运算符**new**为之分配空间后，才可以引用数组中的每个元素。
- 三种初始化：动态初始化（数组定义与为数组元素分配空间并赋值的操作分开进行）；静态初始化（定义赋值同时进行，**int c[]={...}**）；默认初始化。



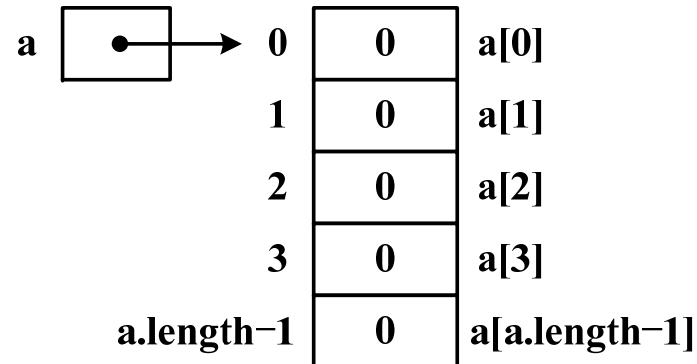
3.6.1 一维数组

数组变量

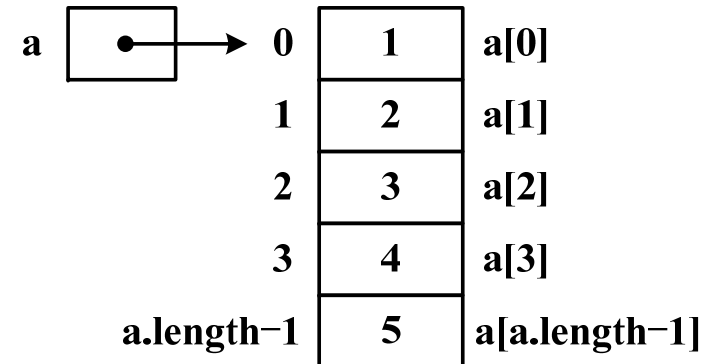
a 未初始化

`int a[];`
(a) 声明数组变量

下标 数组元素



`a=new int [5];`
(b) 申请数组存储空间 (有默认初值)



`for (int i=0; i<a.length; i++)`
`a[i]=i+1;`
(c) 对数组元素进行运算



for语句作用于数组的逐元循环

for (类型 变量 : 数组)

for (int value : fib)

//value获得**fib**数组每个元素,

//相当于fib[i]

System.out.print(" "+value);

数组的引用模型

① 基本数据类型变量的传值赋值

i	10
j	未初始化

int i=10, j;

(a) 变量声明

i	10
j	10

j=i;

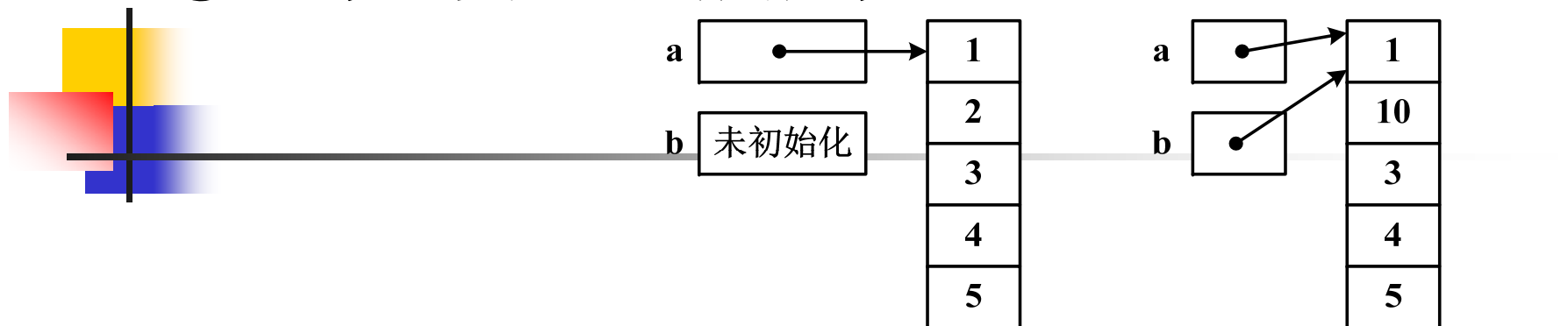
(b) 变量赋值，传值

i	10
j	11

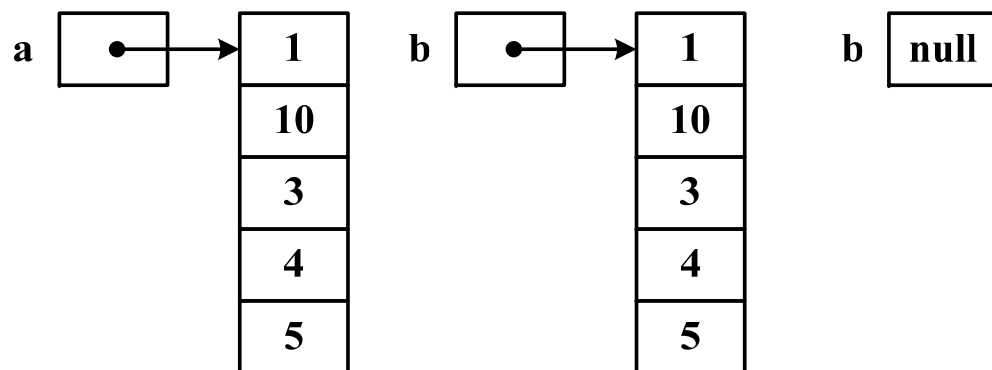
j++;

(c) 对一个变量赋值不影响其他变量

② 数组变量的引用赋值



(a) `int a[]={1,2,3,4,5}, b[];` `b=a;` //引用赋值
`b[1]=10;`
 (b) 数组变量赋值, 传递引用,
`a==b`结果为true, `a[1]==10`



`b=new int [a.length];`
`for (int i=0; i<a.length; i++)`
`b[i]=a[i];`
 (c) `b`重新申请数组空间, 复制`a`数组
 元素到`b`, `a==b`结果为false

`b=null;`
 (d) 释放数组
 占用的存储空间



3.6.2 二维数组

1. 声明二维数组

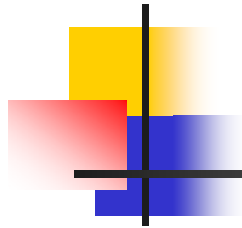
```
int mat[][] = new int [3][4];
```

```
int mat[][] = { {1,2,3},{4,5,6} };
```

二维数组元素表示格式如下：

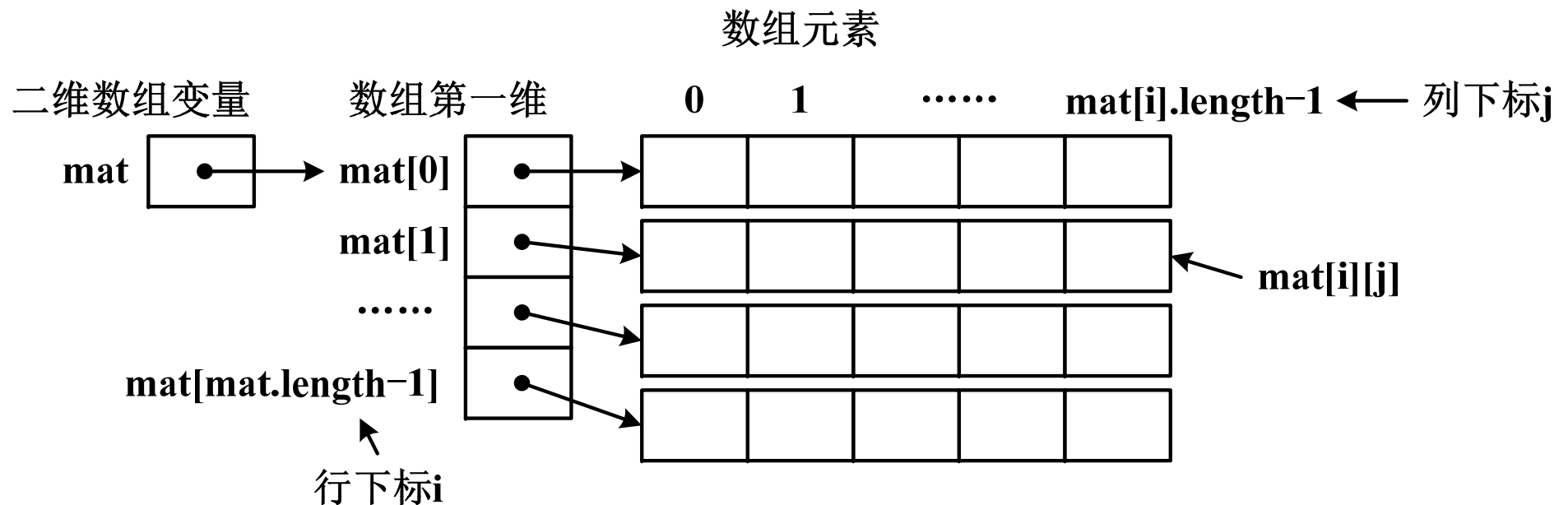
二维数组[下标**1**][下标**2**]

mat[*i*][*j*] //表示第*i*行第*j*列的数组元素

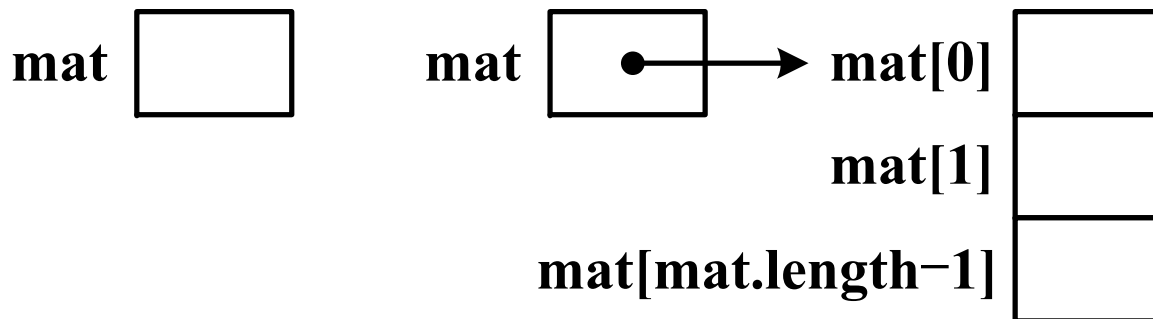


二维数组的引用模型

```
int mat[][] = new int [m][n];
```

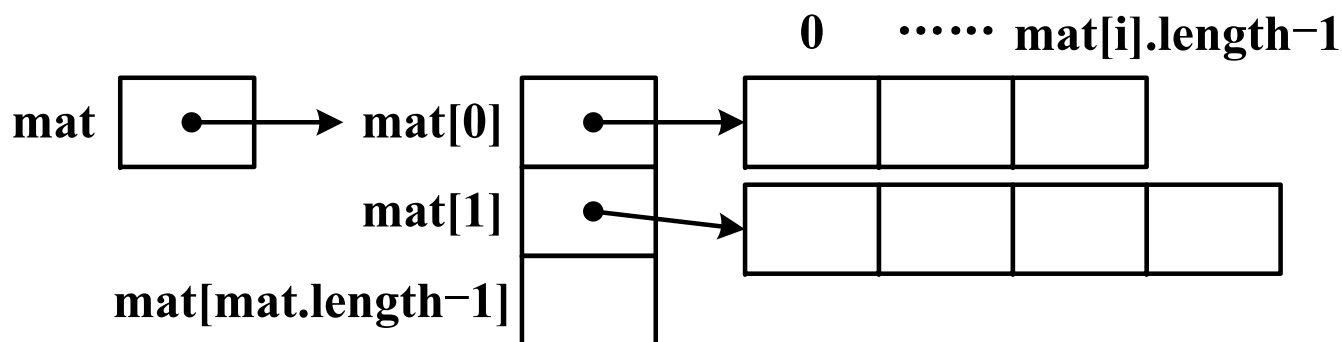


不规则的二维数组



(a) `int mat[][];`

(b) `mat=new int [3][];`



(c) `mat[0]=new int [3];`
`mat[1]=new int [4];`



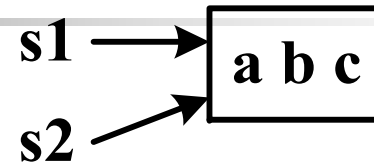
3.7 字符串

String类又称为不可变字符序列。

String类位于**java.lang**包中，**Java**程序默认导入**java.lang**包下的所有类。

Java字符串就是**Unicode**字符序列，例如字符串“**Java**”就是**4**个**Unicode**字符'**J**'、'**a**'、'**v**'、'**a**'组成的。

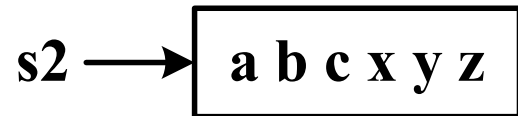
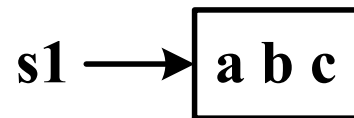
字符串的引用模型



```
String s1 = "abc";
```

```
String s2 = s1;    //引用赋值
```

(a) 字符串变量赋值，传递引用



```
s2 += "xyz";
```

(b) 再对s2赋值，重新分配存储空间，不影响s1字符串



字符串的类特性

字符串变量.**方法**([参数列表])

int week=1;

String str="日一二三四五六";

//每汉字的字符长度为1

"星期"+str.charAt(week) //charAt(1)获得字符'一'

“星期”+str.substring(week, week+1)

//substring(1,2)获得子串 “一”

str.length()

//获得str的长度

附录：类、变量、方法、接口修饰符总结

(1) 类:

访问修饰符 修饰符 class 类名称 extends 父类名称（单个） implement 接口名称（可以多个）（访问修饰符与修饰符的位置可以互换）。

访问修饰符:

名称	说明	备注
public	可以被所有类使用	public类必须定义在和类名相同的同名文件中，且一个java文件中只能最多有一个public类
缺省	可以被同一个包中的类使用	默认访问权限，可以省略此关键字，可以定义在和public类同一个文件中

修饰符:

名称	说明	备注
final	使用此修饰符的类不能被继承	
abstract	如果要使用abstract类，之前必须要建一个继承abstract类的子类，子类中实现abstract类中的抽象方法 abstract类不能实例化对象	类只要有一个abstract方法，类就必须定义为abstract，但abstract类不一定非要包括abstract方法。



附录：类、变量、方法、接口修饰符总结

(2) 变量：

- **Java**中没有全局变量，只有局部变量、实例变量、类变量（类中的静态变量）。
- 方法中的变量不能够有访问修饰符。访问修饰符仅针对类中定义的变量。
- 声明实例变量时，如果没有赋初值，将被实例化为**null**（引用类型）或**0**、**false**（原始类型）。
- 可以通过实例变量初始化块（用**{}**包含的语句块）来初始化较复杂的实例变量，在类的构造器被调用时运行，运行时刻位于父类构造器之后，本类构造器之前。
- 类变量（静态变量）也可以通过类变量初始化块（用**static{}**包含的语句块）来进行初始化，且该静态块在所有层级上的构造函数调用之前被调用。只可能被初始化一次。
- **static{}**是在类初始化时执行，不是在创建对象时执行。静态初始化块中不能访问非**static**成员变量。执行顺序：上溯到**Object**类，先执行**Object**的静态初始化块，再向下执行子类的静态初始化块，直到我们的类的静态初始化块为止。



附录：类、变量、方法、接口修饰符总结

(2) 变量：

访问修饰符：

名称	说明	备注
public	可以被任何类使用	
protected	可以被同一包中的所有类访问，可以被所有子类访问	子类不在同一个包中也可以访问
缺省	可以被同一包中的所有类访问	如果子类不在同一个包中，也不能访问
private	只能被同一类访问	

修饰符：

名称	说明	备注
static	静态变量（类变量）	可以被类的所有实例共享，并不需要创建类的实例就可以访问，在方法区中
final	常量，值只能分配一次，不能更改	注意不要使用const，虽然它和C、C++中的const关键字含义一样。可以同static一起使用，避免对类的每个实例维护一个拷贝
transient	告诉编译器，在类对象序列化的时候，此变量不需要持久保存	该变量可以通过其它变量来得到，使用它是为了性能问题
volatile	指出可能有多个线程修改此变量，要求编译器优化以保证对此变量的修改能正确被处理	



附录：类、变量、方法、接口修饰符总结

(3) 方法：

访问修饰符 修饰符 返回类型 方法名称（参数列表） **throws Exception**列表（访问修饰符与修饰符的位置可以互换）。

- 类的构造器不能够有修饰符、返回类型和**throws**子句；
- 类的构造器方法被调用时，它首先调用父类的构造器，然后调用实例变量初始化块（**{}**），然后运行构造器本身（会沿类层级迭代进行此步骤）；
- 如果构造器方法没有显式的调用父类的构造器，那么编译器会自动为它加上一个默认的**super()**，而如果父类又没有默认的无参数构造器，则编译器报错。**super**必须是构造器方法的第一个子句。
- 同一个构造方法里面不能同时调用**super()**和**this()**。



附录：类、变量、方法、接口修饰符总结

(3) 方法：

访问修饰符：

名称	说明	备注
public	可以从所有类访问	
protected	可以被同一包中所有类访问，可以被所有子类访问	子类没有在同一包中也可以访问
缺省	可以被同一包中的所有类访问	如果子类没有在同一包中，也不能访问
private	只能够被当前类的方法访问	



附录：类、变量、方法、接口修饰符总结

(3) 方法：

修饰符：

名称	说明	备注
static	静态方法（类方法）	提供不依赖于实例对象的服务，并不需要创建类的实例就可以访问静态方法
final	防止任何子类重写该方法，但是可以重载该方法	注意不要使用const，虽然它和C、C++中的const关键字含义一样。可以同static一起使用，避免对类的每个实例维护一个拷贝
abstract	抽象方法，类中已声明而没有实现的方法	不能将static方法、final方法或者类的构造器方法声明为abstract
native	用该修饰符定义的方法在类中没有实现，而大多数情况下该方法的实现是用C、C++编写的	参见Java Native接口（JNI），JNI提供了运行时加载一个native方法的实现，并将其与一个java类关联的功能
synchronized	多线程的支持	当一个此方法被调用时，没有其它线程能够调用该方法，其它的synchronized方法也不能调用该方法，直到该方法执行完成



附录：类、变量、方法、接口修饰符总结

(4) 接口：

访问修饰符 **interface** 接口名称 **extends** 接口列表。

- 接口不能够定义其声明的方法的任何实现；
- 接口中所有方法默认隐含为**public**（只能）的；
- 接口中不能够定义变量，接口中的变量总是需要定义为“**public static final** 接口名称”，但可以不包含这些修饰符，编译器默认就是这样，显式的包含修饰符是为了程序清晰。

访问修饰符：

名称	说明	备注
public	所有类可见	
缺省	同一个包内	