

《面向对象设计与构造》

Lec05-对象并发及其协同

OO课程组2022

北京航空航天大学计算机学院

提纲

- 单元主题分析
- Java程序是如何运行的
- 线程程序
- 线程状态与调度机制
- 线程交互与对象交互
- 几种线程交互模式
- 本周作业解析

单元主题分析

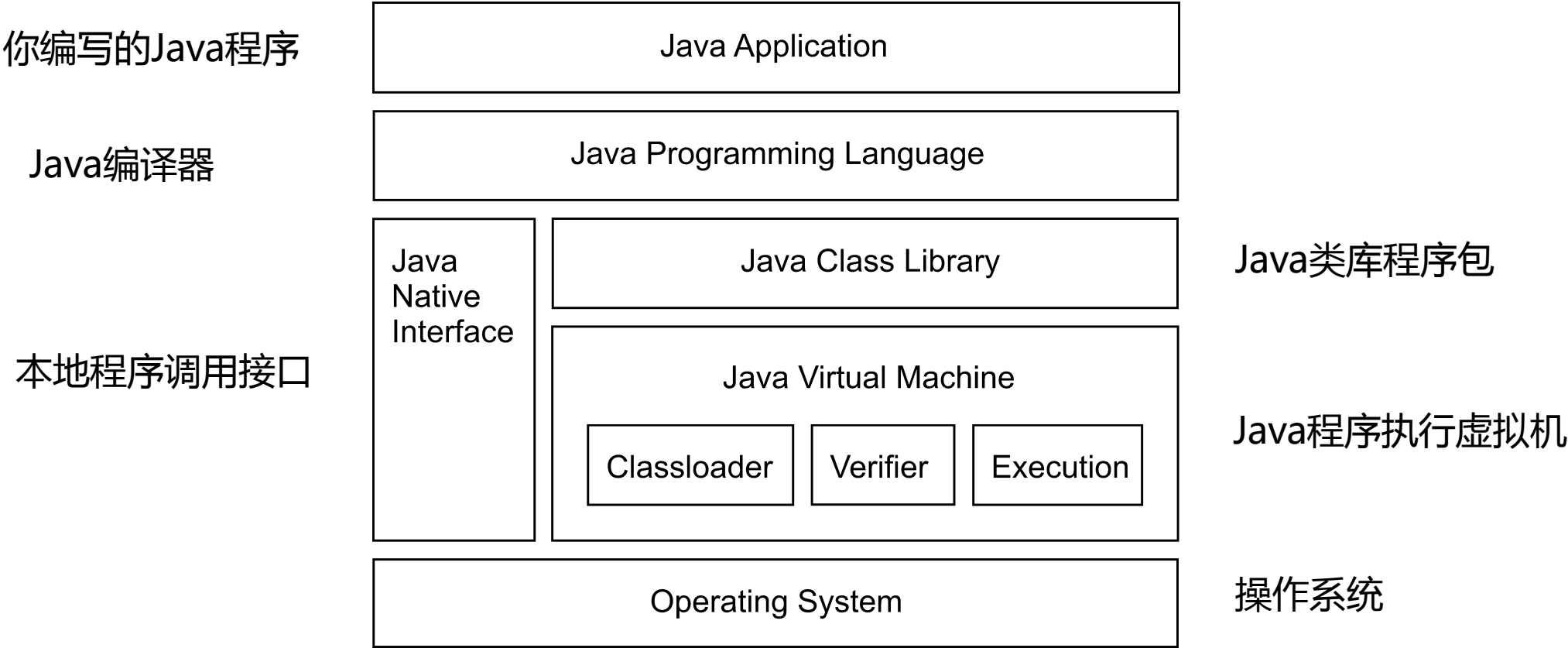
- 对象行为之间具有一定的关系
 - 顺序关系：a.m1();b.m2();
 - 层次代理关系：a.m1()→b.m2();
 - 控制协调关系：a.m1()→{b.m2();if(...)c.m3();d.m4();...}
 - 数据依赖关系：a.m1()-->b-->c.m3()
- 在串行执行的程序中
 - 任何时候：只有一个对象在执行→只有一个对象的数据在被读或写
- 如果程序允许多个对象并发执行呢？

```
a. m1 () ; | x. f () |  
b. m2 () ;
```

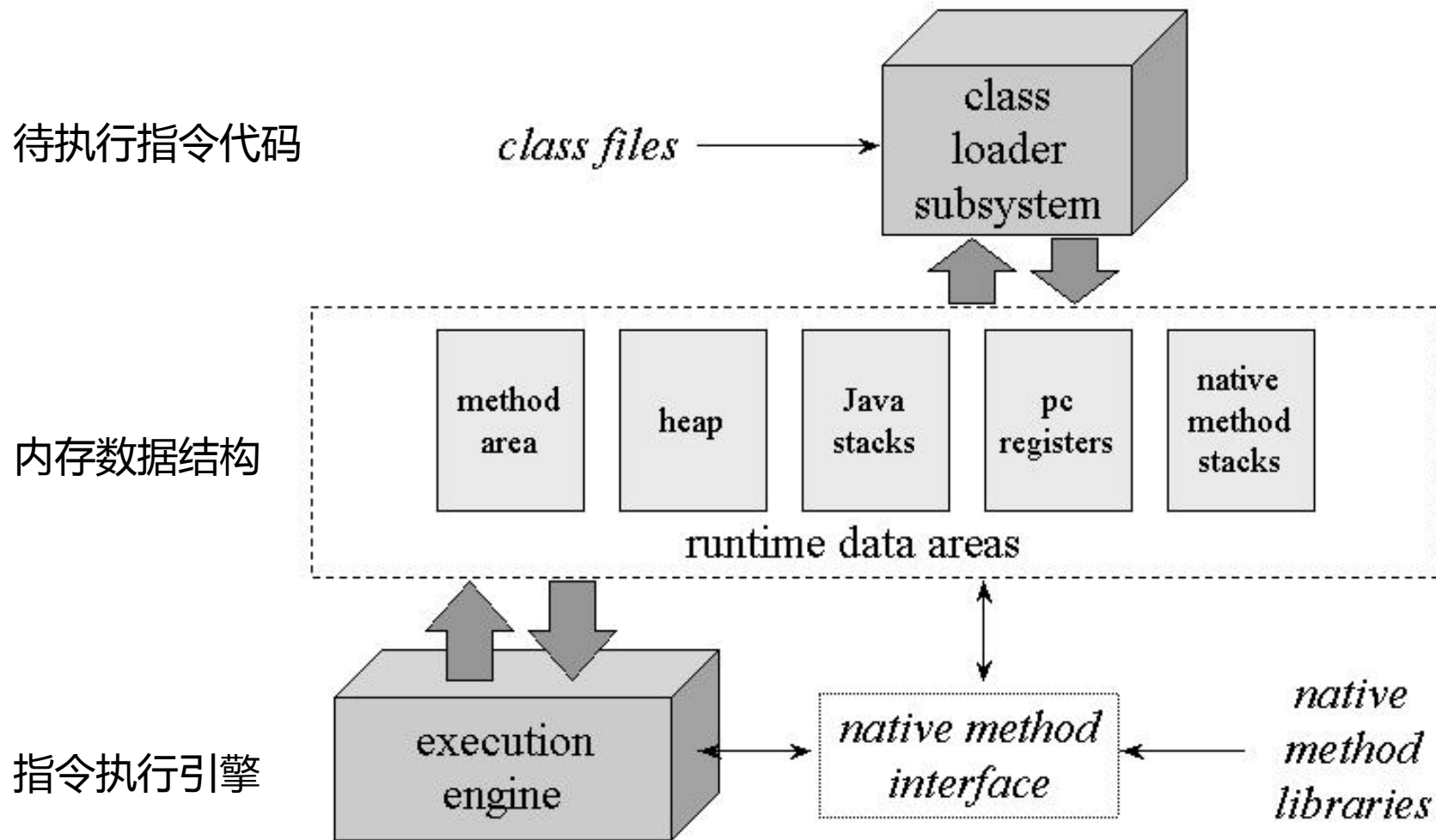
```
a. m1 () --> b | x. f () --> b |  
c. m3 <-- b
```

- Java程序的运行时系统原生支持了对象并发执行

Java程序的运行时系统

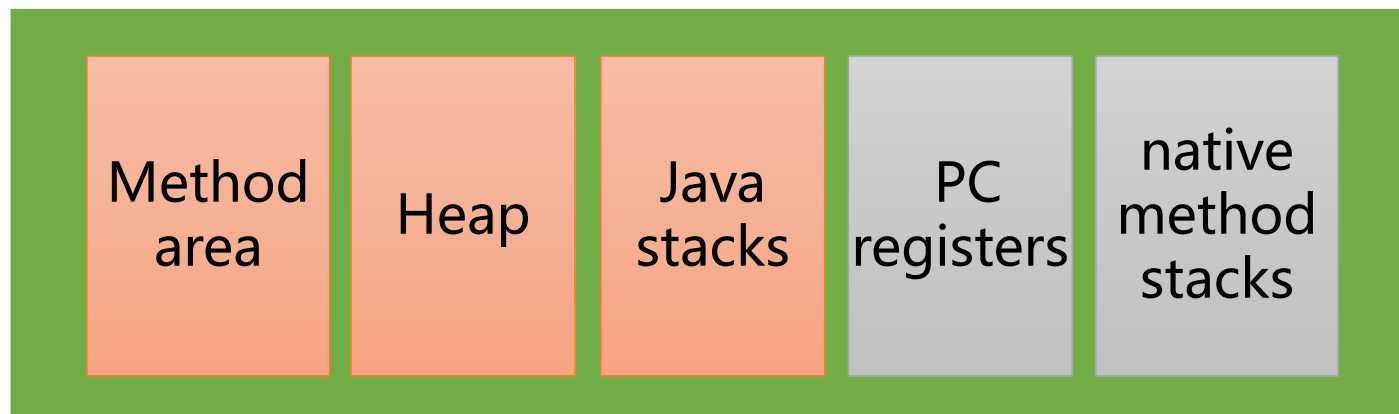


Java程序在JVM中执行



JVM内存区域划分

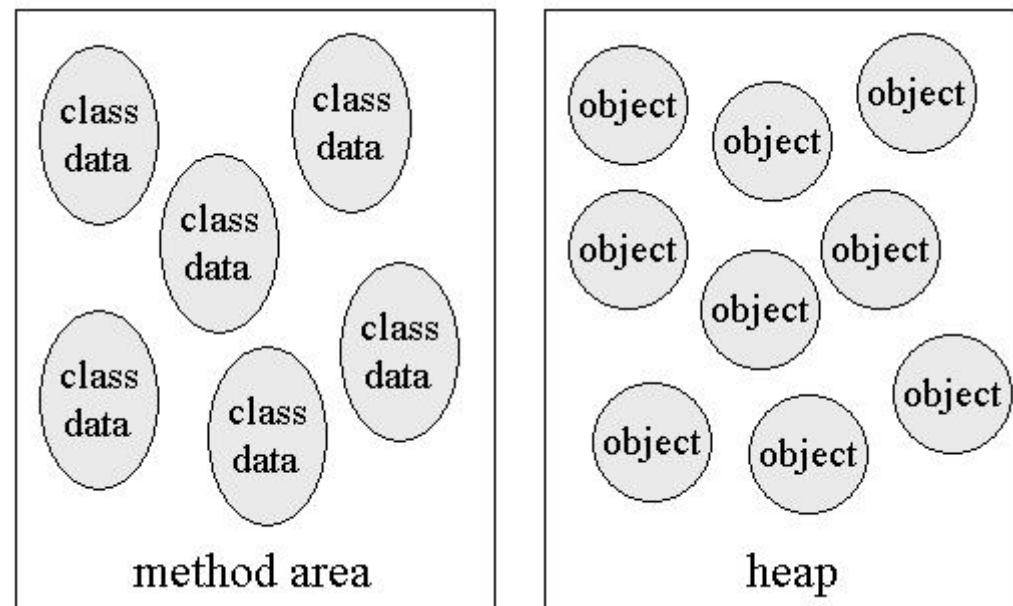
- Java程序运行时由一到多个线程组成
- 每个线程都有自己的栈(**局部栈**)
- 所有线程共享同一个堆(**全局堆**)



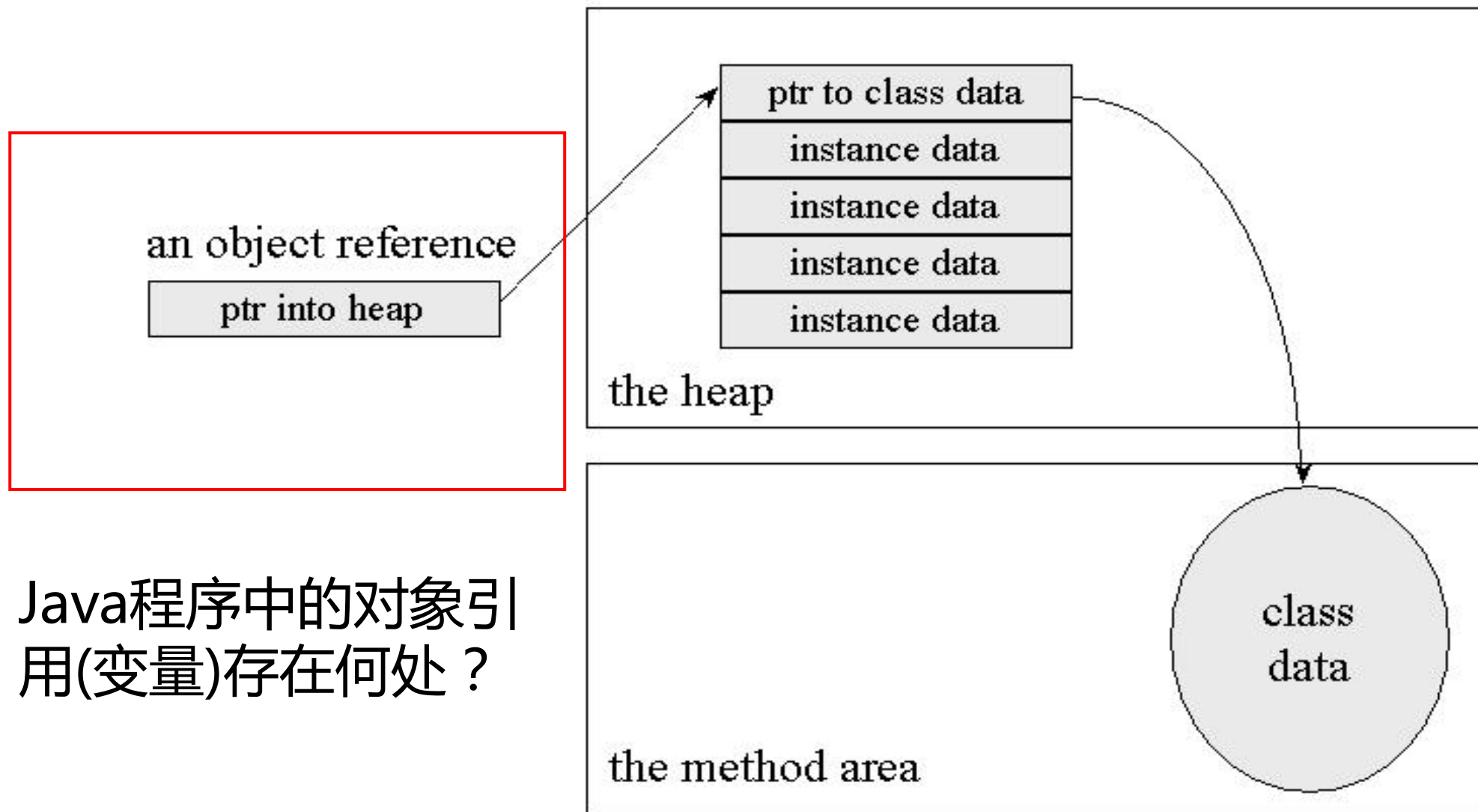
- Method area
 - Class description
 - Code/Instruction
 - Constant pool
- Heap
 - Objects and Arrays
 - Shared by all threads
 - Garbage collection
- Stack
 - Invocation frame
 - Local variable area
 - Operand stack

JVM内存区域划分

- Method内存区域
 - 保存class信息(类型定义信息)
 - 每个Java应用拥有一个相应区域
 - 虚拟机中的所有线程共享
 - **一次只能由一个线程访问**
- Heap内存区域
 - 保存对象或数组
 - 每个Java应用拥有一个相应区域
 - 为垃圾回收提供支持
 - 程序执行过程中动态扩展和收缩



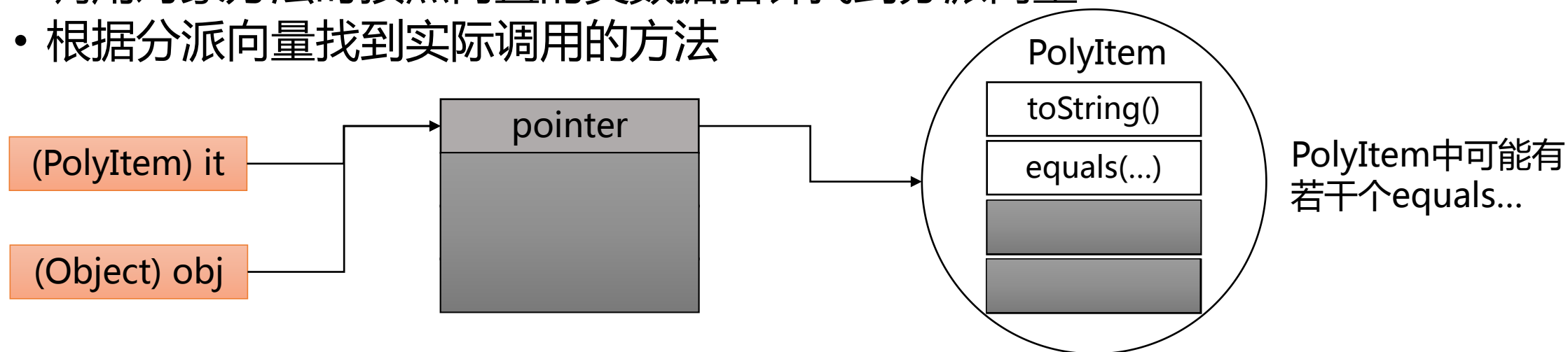
连接起来才能工作



Java程序中的对象引用(变量)存在何处？

如何知道调用哪个方法？

- 类提供了一个列出所有方法的列表：分派向量(dispatch vector)
 - 给出了类中所有方法入口，存放在Method内存区的class数据中
- 每个对象在创建时按照创建类型内置一个指向类数据的指针
 - 对象引用类型的变化不会改变对象的创建类型和对象内容
 - 调用对象方法时按照内置的类数据指针找到分派向量
 - 根据分派向量找到实际调用的方法

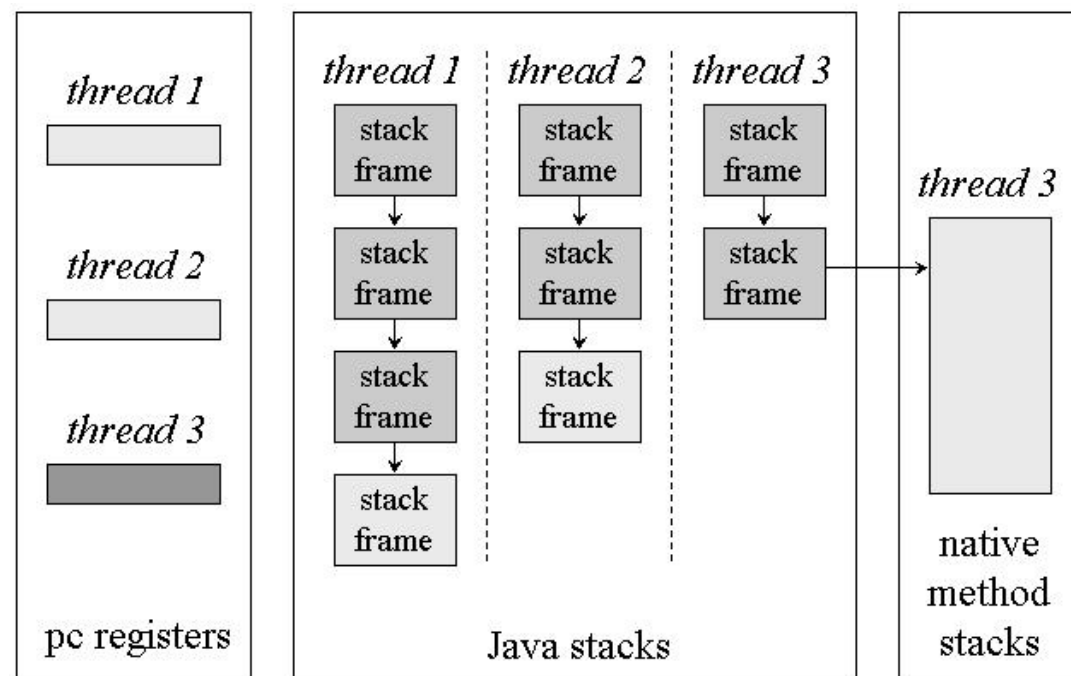


JVM栈的内存结构

- 栈内存区

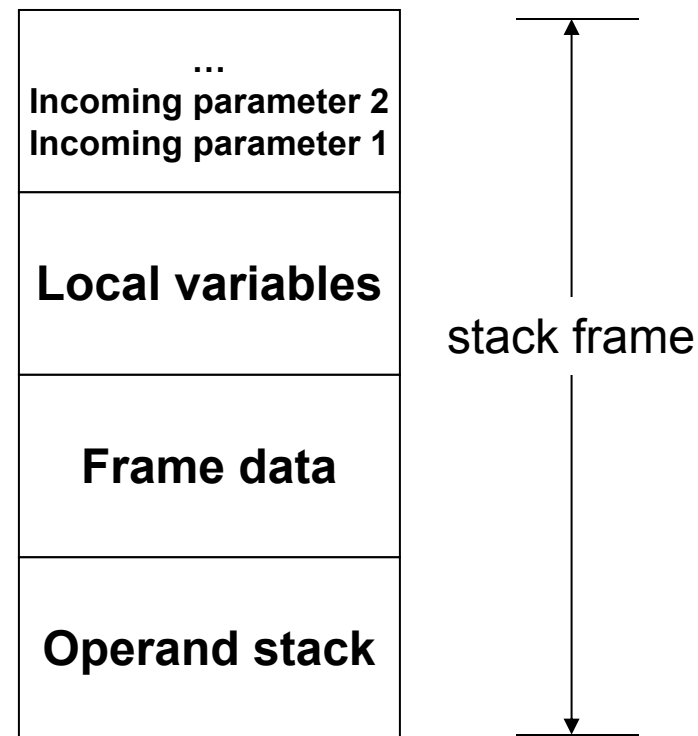
- 每个线程都拥有专属的栈内存，用以追踪方法的执行
- 栈由栈帧组成，顶栈帧描述线程当前在执行哪个方法
- JVM对栈帧进行push和pop操作
 - (程序中)调用方法: (JVM)push
 - (程序中)方法返回: (JVM)pop

线程栈底存的是什么？



栈帧结构

- 方法输入参数
- 方法局部变量
- 栈帧数据(Frame Data)
 - 到类定义区(method area)的引用
 - 方法调用返回值
 - 异常处理入口
- 操作数栈(Operand Stack)
 - 计算操作的工作空间



Java程序运行时的内存状态变化

```
public class BankAccount {  
    private double balance;  
    private static int totalAccounts = 0;  
    public BankAccount() {  
        balance = 0;  
        totalAccounts++;  
    }  
    public void deposit( double amount ) {  
        balance += amount;  
    }  
}
```

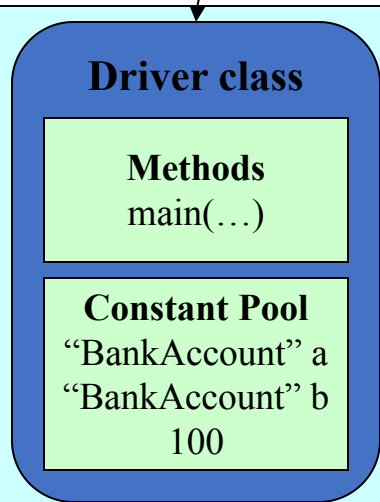
```
public class Driver {  
    public static void main( String[] args ) {  
        BankAccount a = new BankAccount();  
        BankAccount b = new BankAccount();  
        b.deposit( 100 );  
    }  
}
```

```
// In command prompt
java Driver

// In Java Virtual Machine
Driver.main( args )
```

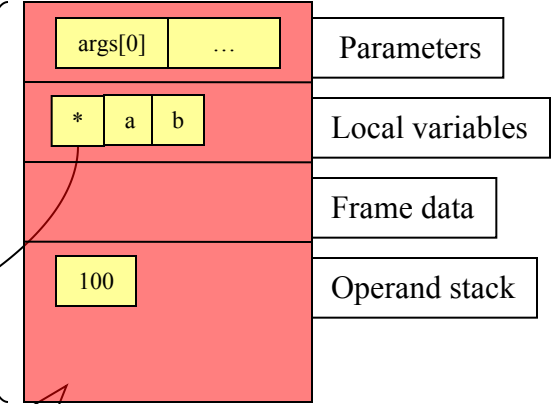
```
public class Driver {
    public static void main( String[] args ) {
        BankAccount a = new BankAccount();
        BankAccount b = new BankAccount();
        b.deposit( 100 );
    }
}
```

ClassLoader把
Driver类加载到方
法区域



Method Area

main()

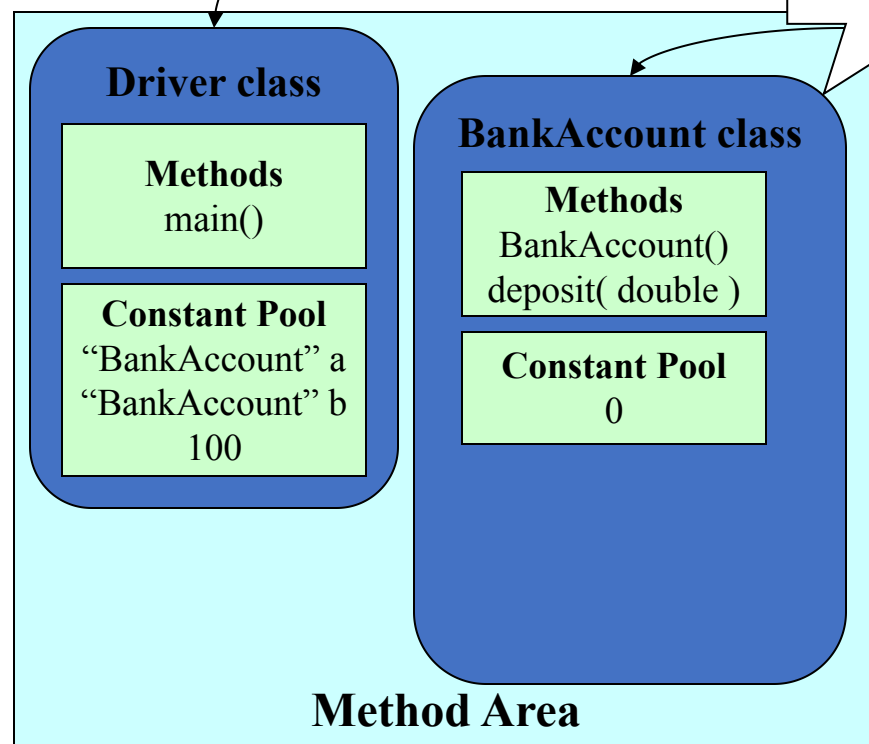


针对main()方法的栈帧
被创建，并处于栈顶。
*为对当前对象对应
class的引用。



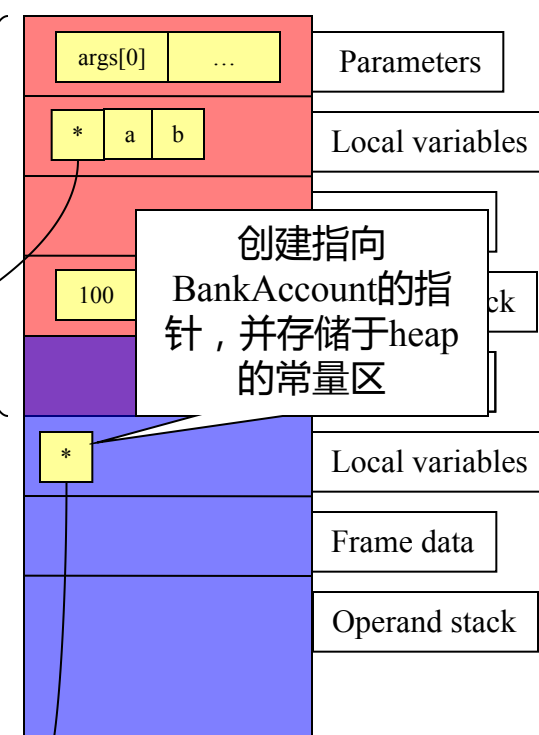
Heap

```
// Driver.java
public static void main( String[] args ) {
    BankAccount a = new BankAccount();
    BankAccount b = new BankAccount();
    b.deposit( 100 );
}
```



main()

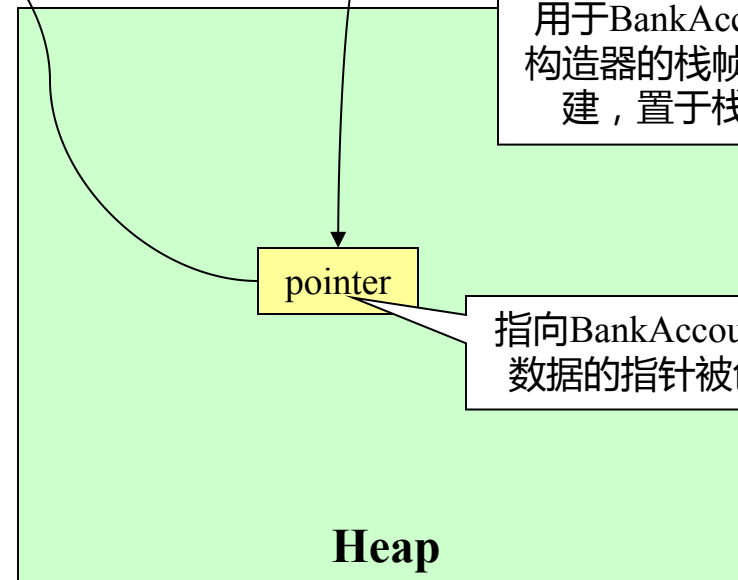
BankAccount()

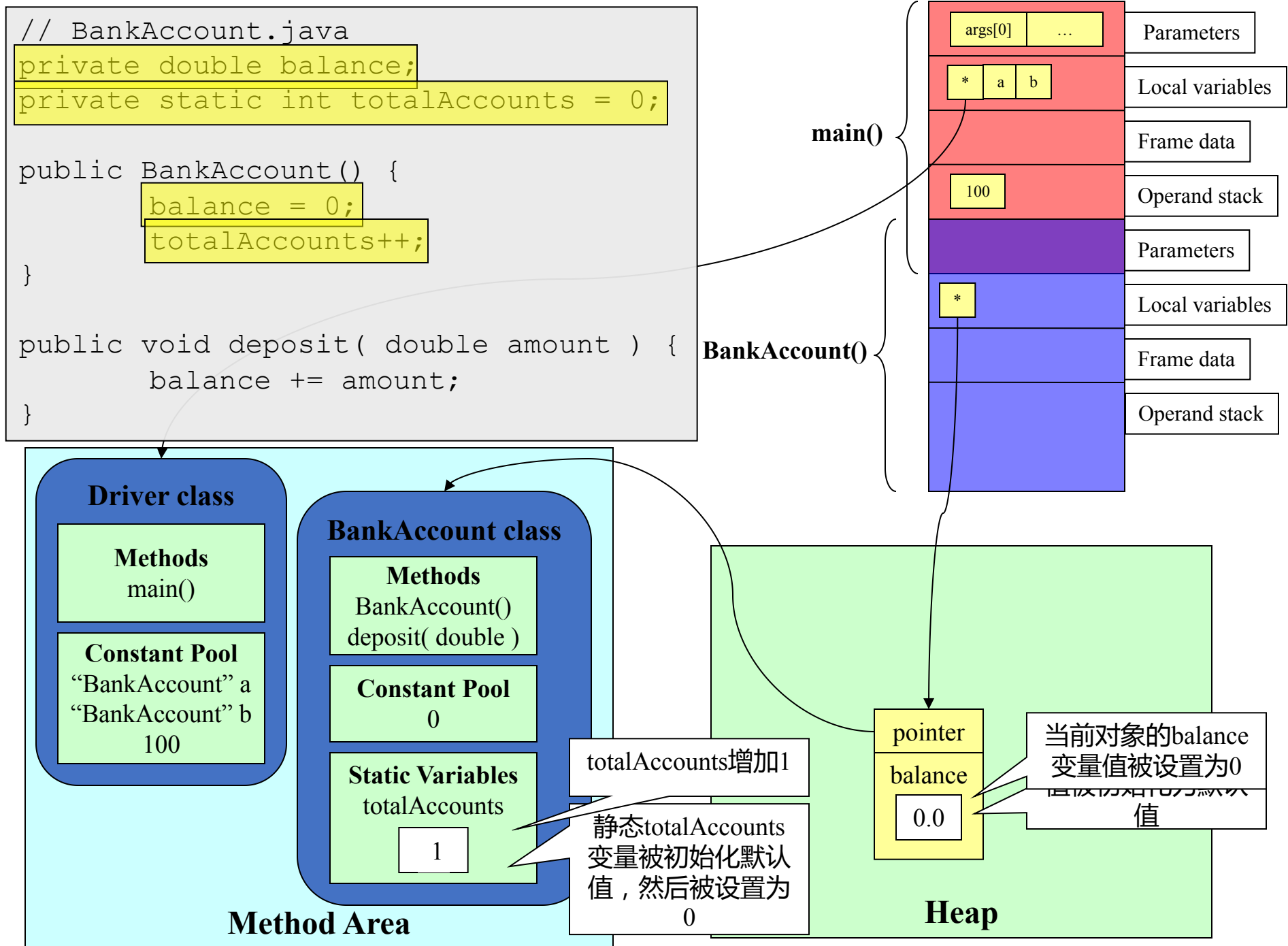


创建指向 BankAccount的指针，并存储于heap的常量区

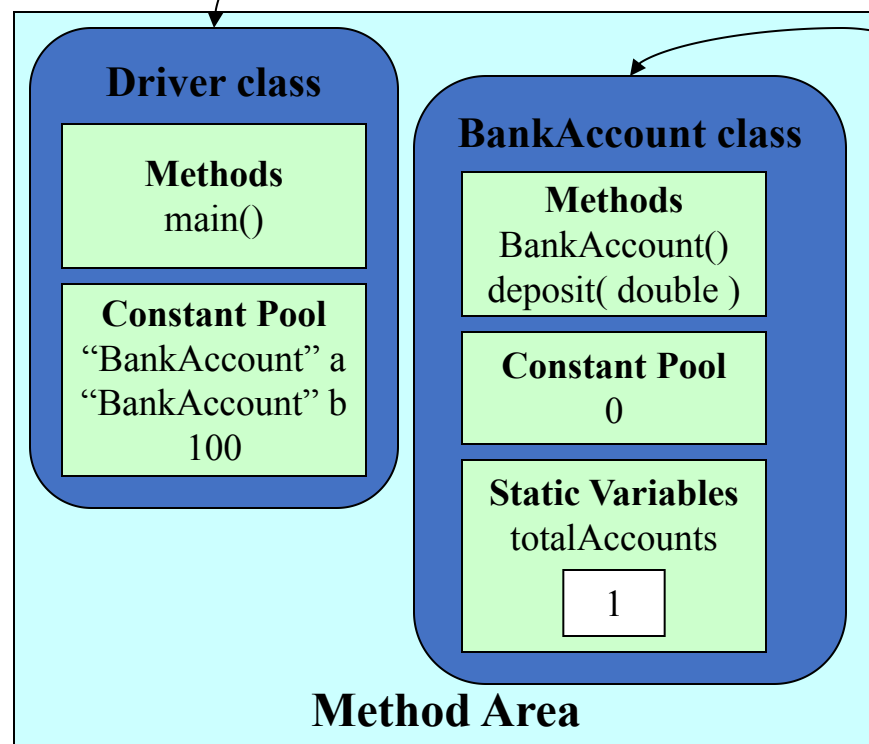
用于BankAccount构造器的栈帧被创建，置于栈顶

指向BankAccount类数据的指针被创建





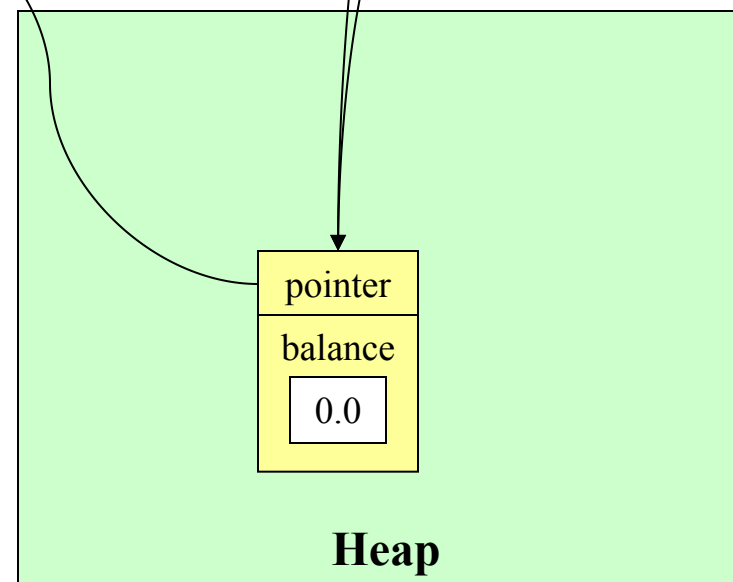
```
// Driver.java
public static void main( String[] args ) {
    BankAccount a = new BankAccount();
    BankAccount b = new BankAccount();
    b.deposit( 100 );
}
```



指针从operand stack弹出，并赋给a

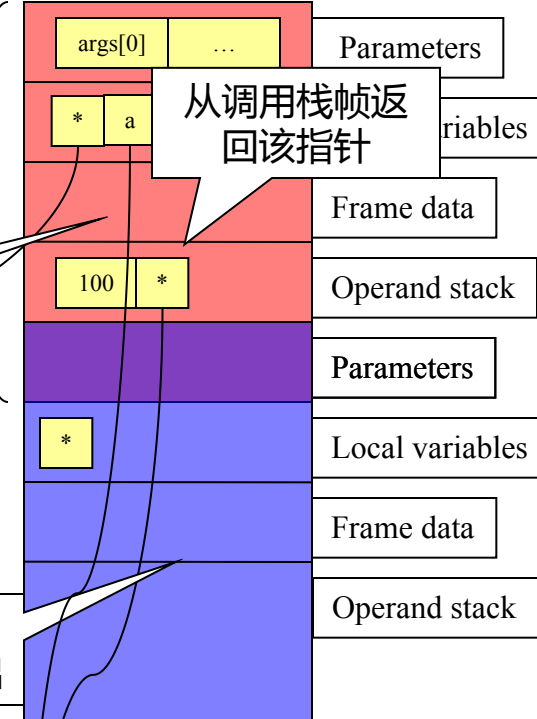
BankAccount()

用于BankAccount构造器的栈帧被弹出



main()

从调用栈返回该指针

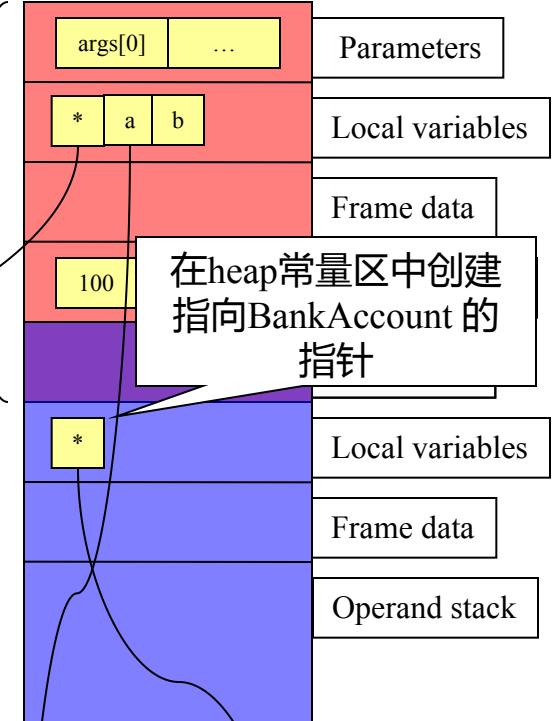



```
// Driver.java
public static void main( String[] args ) {
    BankAccount a = new BankAccount();
    BankAccount b = new BankAccount();
    b.deposit( 100 );
}
```

BankAccount类已经被加载，无需再次加载

BankAccount()

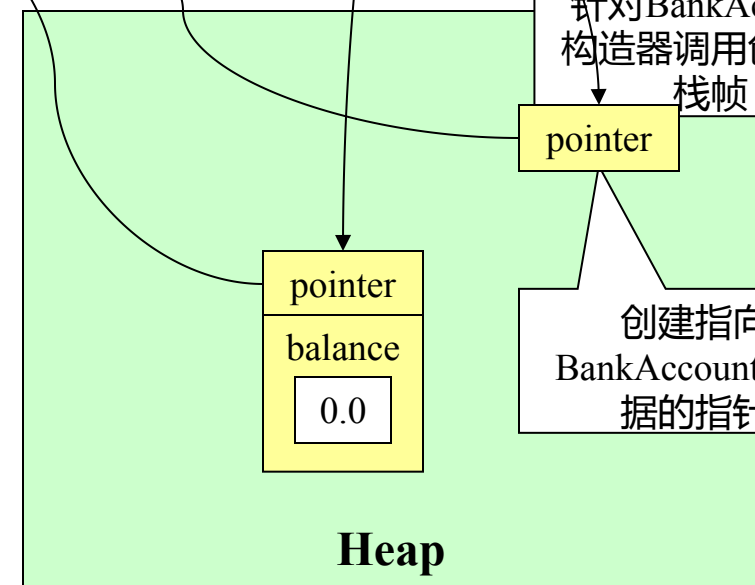
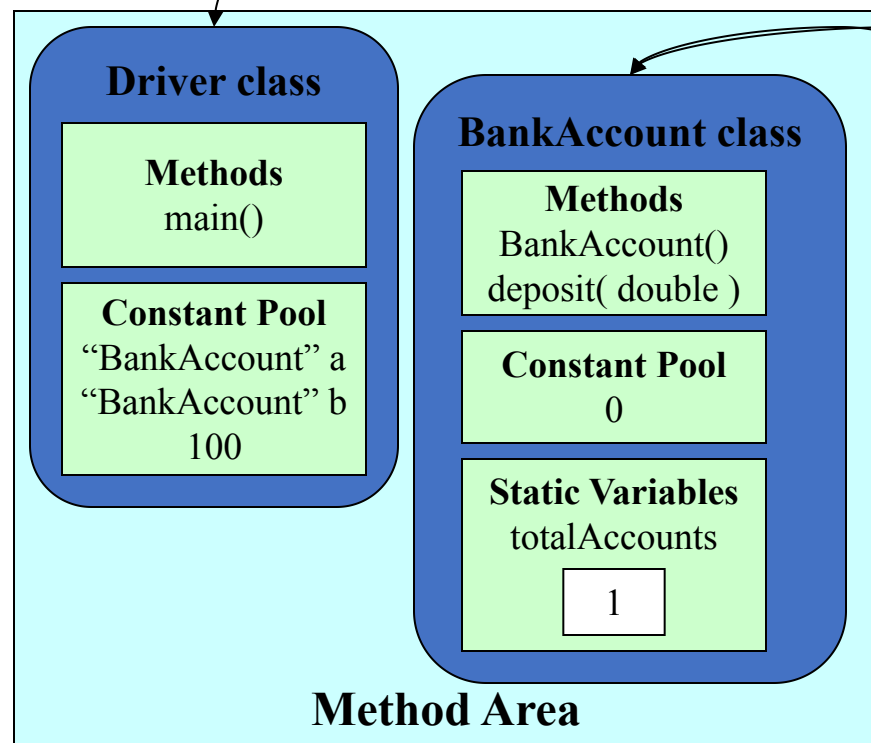
main()

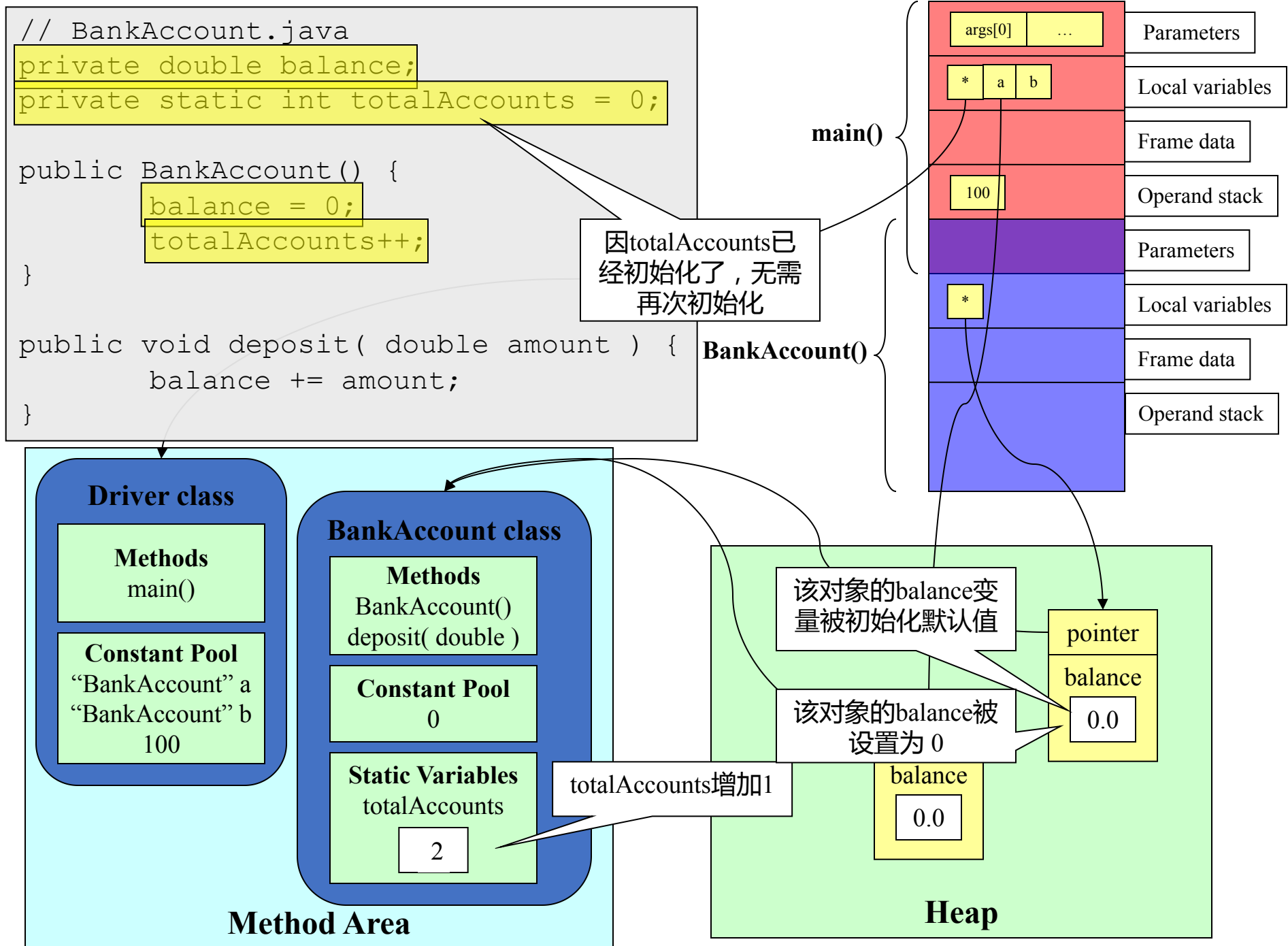


在heap常量区中创建指向BankAccount的指针

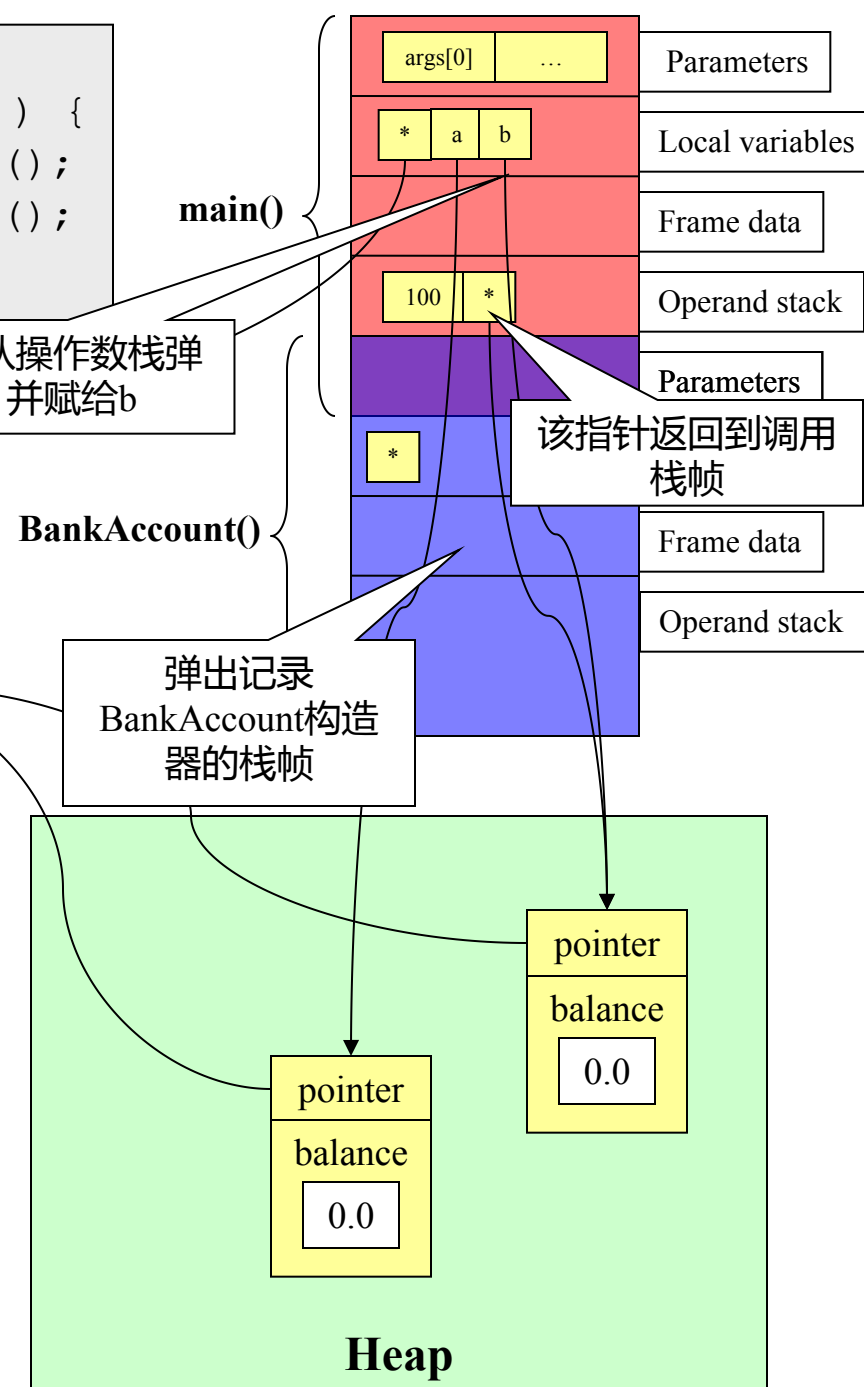
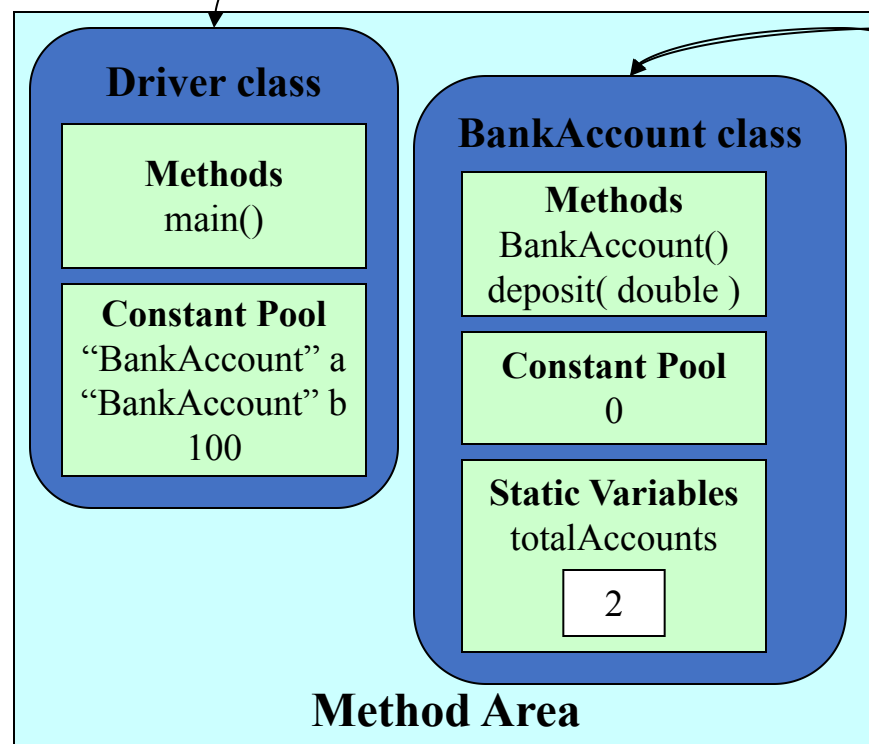
针对BankAccount构造器调用创建的栈帧

创建指向BankAccount类数据的指针

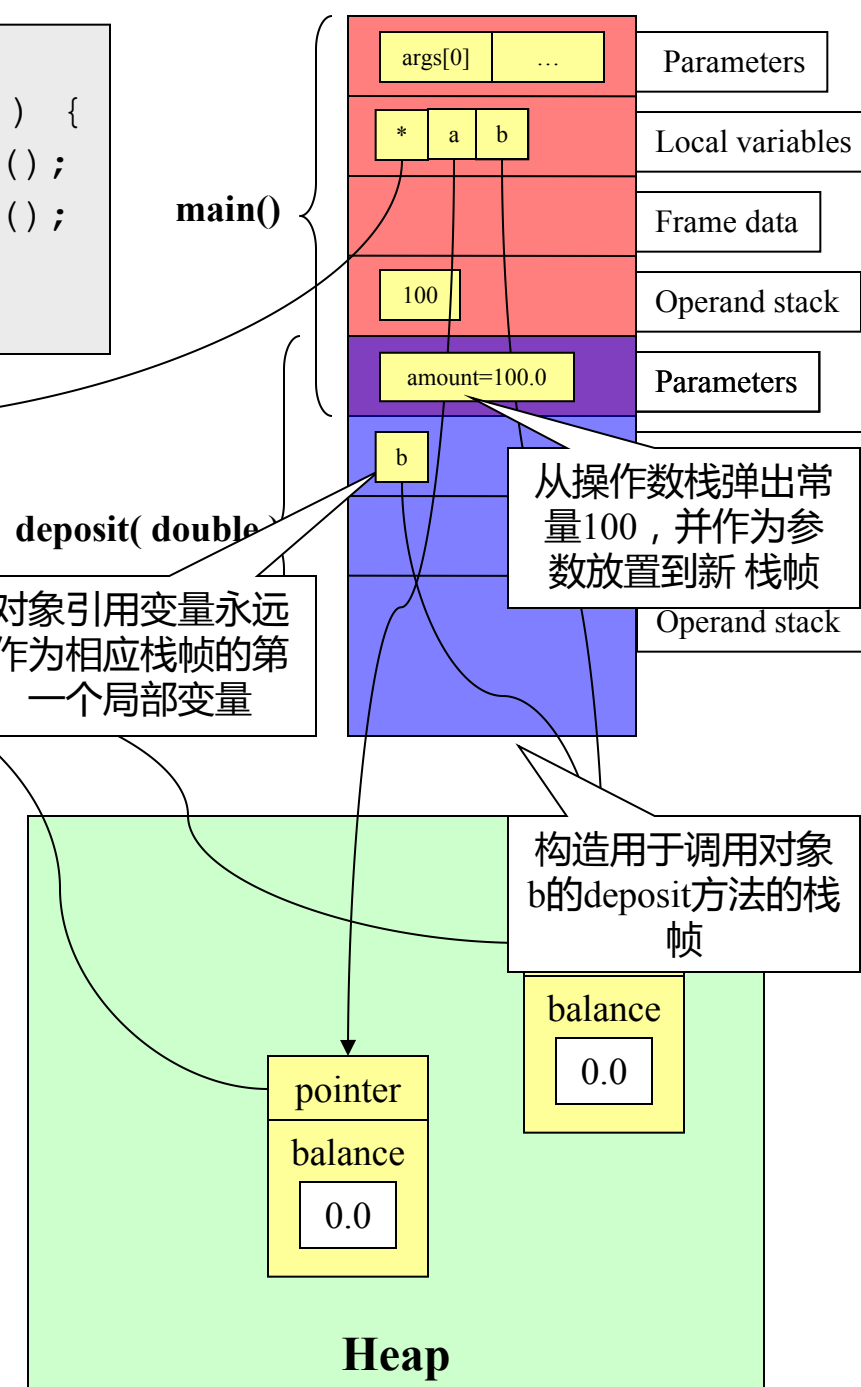
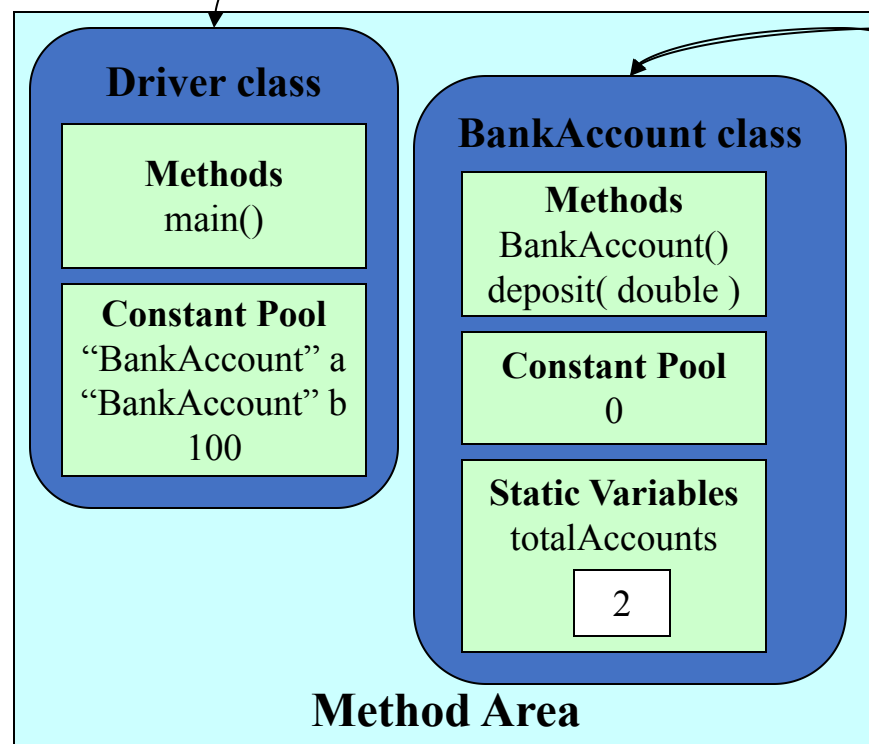




```
// Driver.java
public static void main( String[] args ) {
    BankAccount a = new BankAccount();
    BankAccount b = new BankAccount();
    b.deposit( 100 );
}
```



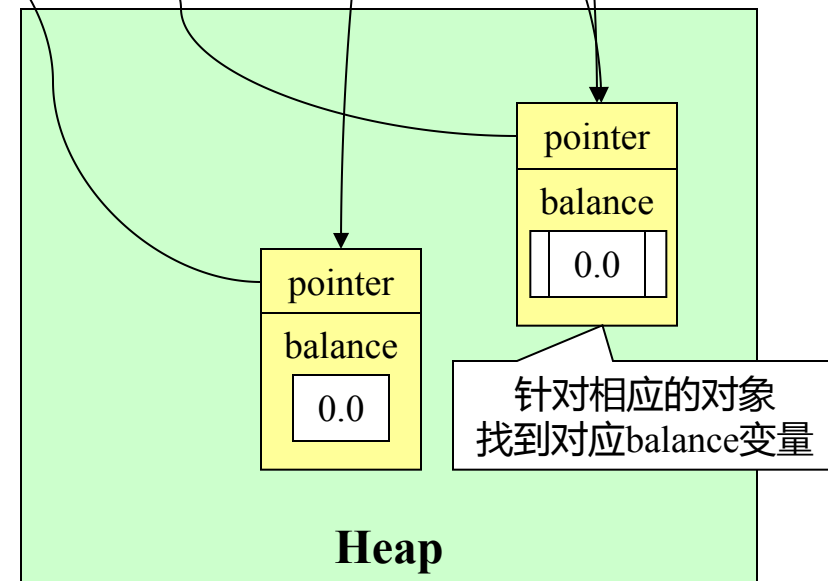
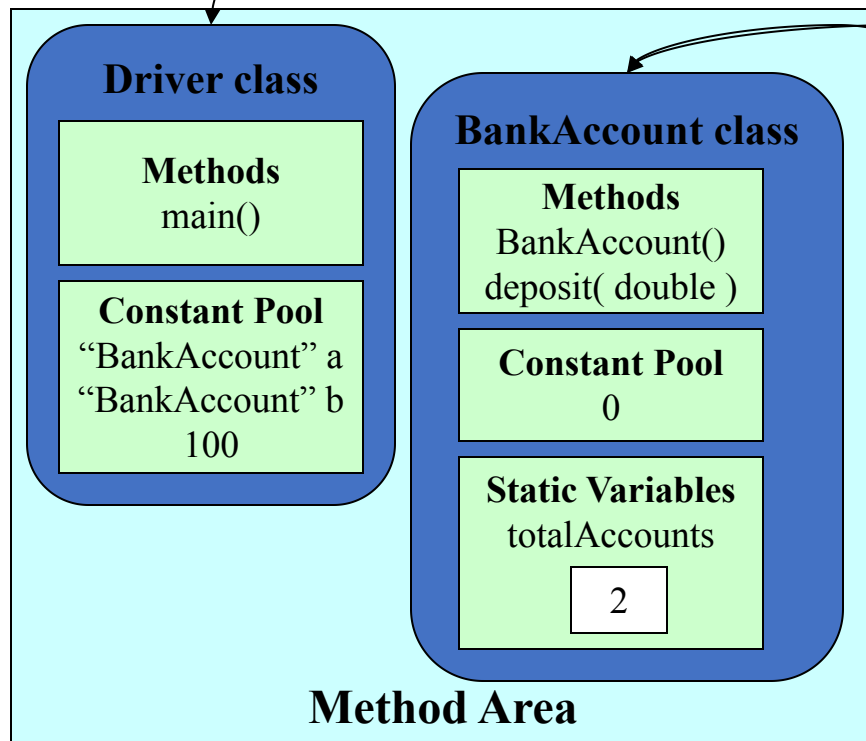
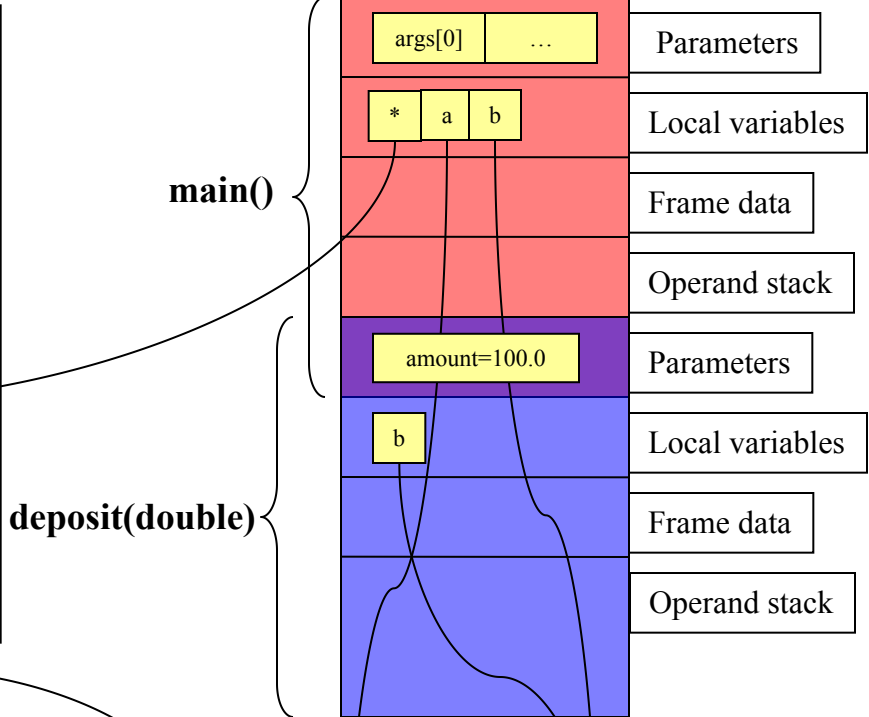
```
// Driver.java
public static void main( String[] args ) {
    BankAccount a = new BankAccount();
    BankAccount b = new BankAccount();
    b.deposit( 100 );
}
```



```
// BankAccount.java
private double balance;
private static int totalAccounts = 0;

public BankAccount() {
    balance = 0;
    totalAccounts++;
}

public void deposit( double amount ) {
    balance += amount;
}
```

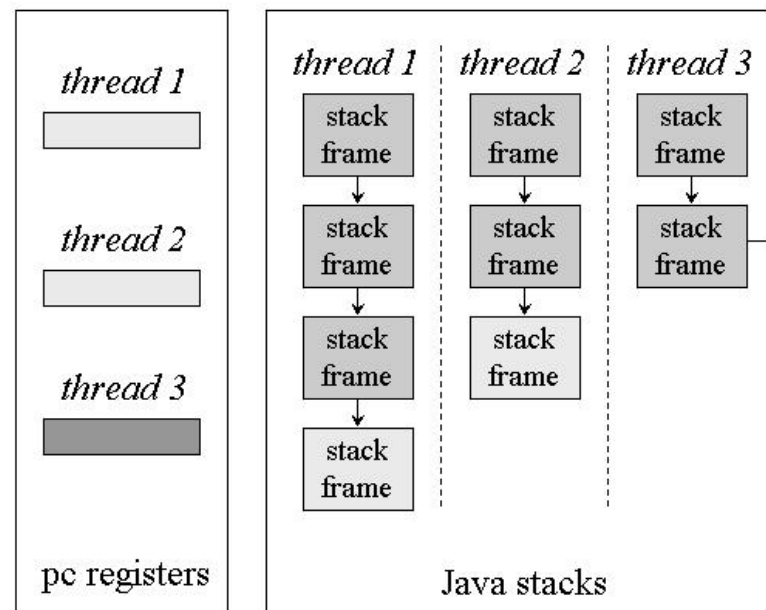
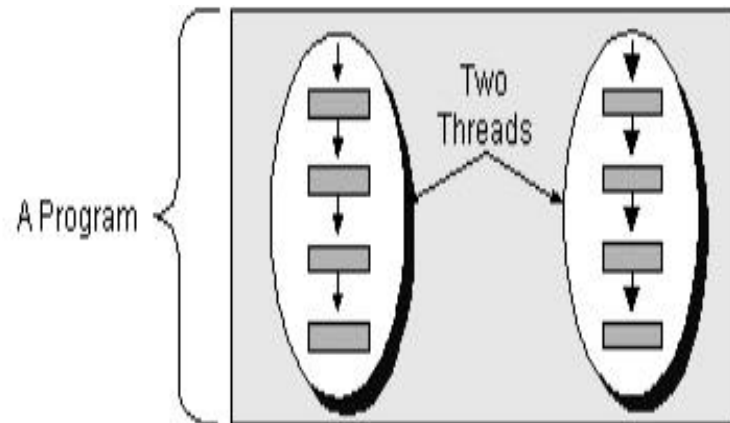


课堂讨论

- 垃圾回收是Java虚拟机的一个核心功能，它的基本工作原理是扫描堆中的对象，记录对象的引用计数，如果发现一个对象不能被程序访问到，就标记为“垃圾”，从而在合适时机回收其内存。
请思考并讨论：
 - 1. 如果堆中某个对象仍然被程序引用，那么可以从JVM的哪个区域找到这些引用？
 - 2. 一个对象的引用计数如果不为0，是否仍然可能被回收？

线程是什么

- 执行(控制)流(flow of control)是程序的一个执行流程
 - 执行流内的指令顺次运行
 - 任意时刻，执行流中只能有一个指令在运行
 - 每个执行流都使用栈来追踪函数调用
- **线程对象**封装了执行(控制)流的管理数据
 - 执行流的**控制数据**：入口地址、返回地址
 - 执行流的**业务数据**：对象引用、变量、常量
- **业务对象**封装了问题域的业务数据



多线程Java程序

- 如何创建线程类
 - 继承Thread类，重写run方法
 - 实现Runnable接口，实现run方法
- 如何创建线程对象
 - Thread t=new TA(...);
 - Thread t = new Thread(new TB(...));
- 如何控制线程对象的执行
 - t.start()
 - t.sleep(...)
 - ...

```
// 继承Thread类
public class TA extends Thread {
    public void run() {    // 执行入口点
        this.go();
    }
}
```

```
// 实现Runnable接口
public class TB implements Runnable {
    public void run() {    // 执行入口点
        this.go();
    }
}
```


线程入口的代码模板

```
public void run() {  
    try { ...  
        while (has work to do) {    // 常规唤醒从这里继续执行  
            do some work;  
            sleep(...) or wait(...); // 休息一会，让其他线程有机会执行  
        }  
    }  
    catch (InterruptedException e) { // 异常唤醒(即interrupt被调用)从这里继续执行  
        ... // 状态不太正常，善后处理...  
    }  
}
```

多个线程的并发执行效果--不确定性

```
public class SimpleThread extends Thread {  
    public SimpleThread(String str) { super(str); }  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(i+ " "+getName());  
            try {  
                sleep((long)(Math.random() * 1000));  
            } catch (InterruptedException e) {}  
        }  
        System.out.println("DONE!" + getName());  
    }  
}
```

```
public class TwoThreadsTest {  
    public static void main (String[] args){  
        new SimpleThread("t1").start();  
        new SimpleThread("t2").start();  
    }  
}
```

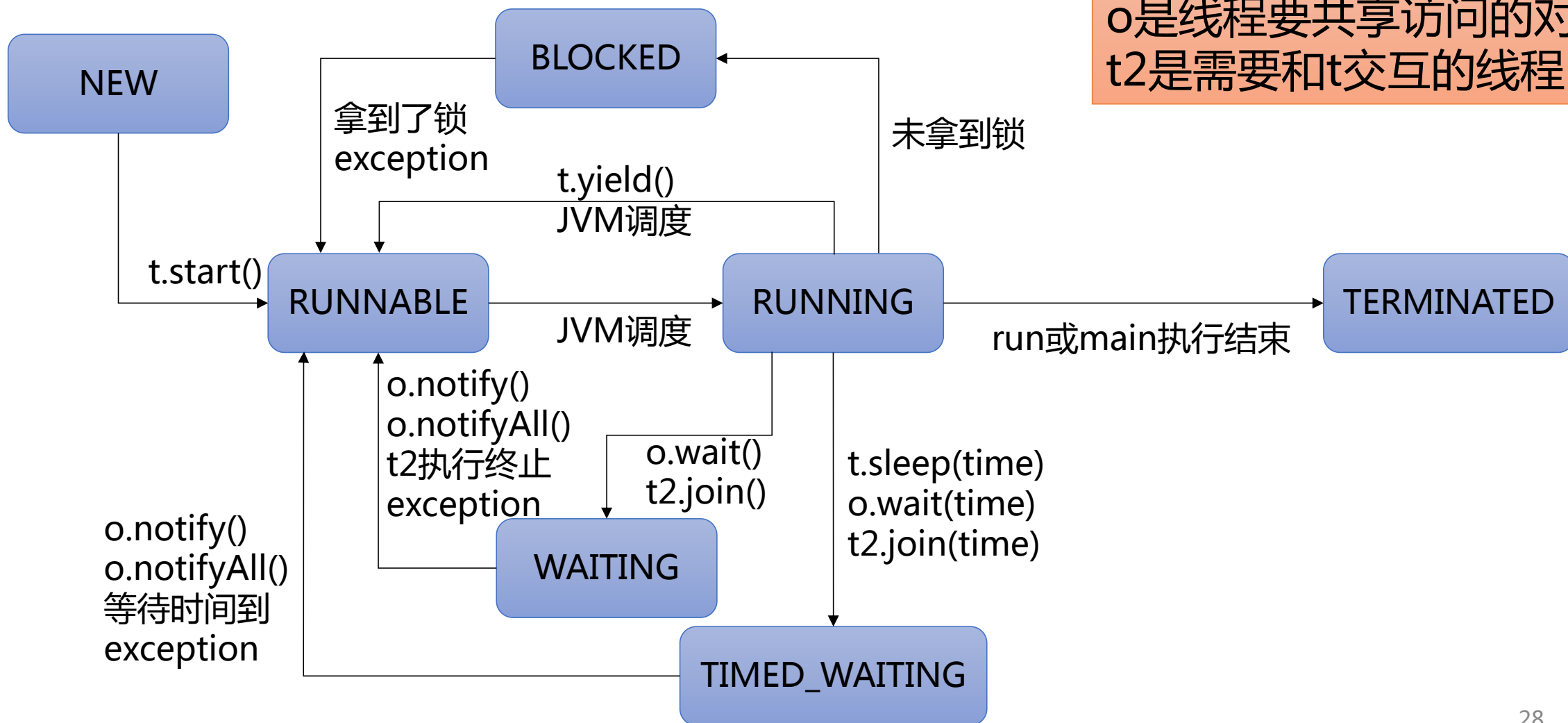
0 t1	4 t1	8 t2
0 t2	4 t2	9 t2
1 t2	5 t1	8 t1
1 t1	5 t2	DONE!
2 t1	6 t2	t2
2 t2	6 t1	9 t1
3 t2	7 t1	DONE!
3 t1	7 t2	t1

线程的运行状态

NEW, // 线程对象被创建后的初始状态
RUNNABLE, // 正运行(running)或准备被调度(ready)
BLOCKED, // 阻塞状态, 无法访问受保护的共享对象
WAITING, // 不定长时间的等待状态
TIMED_WAITING, // 定长时间的等待状态
TERMINATED // run()执行结束或stop()被调用

线程调度下的状态变化

t是当前线程
o是线程要共享访问的对象
t2是需要和t交互的线程



线程调度应与业务结合起来

- 多个线程并发运行，**交换数据**或**共用工具**是业务需要
 - 不知道是否有新数据，或者工具是否可用 → 循环查询
- 如果一直没有新数据或工具不可用呢？
 - 始终处于查询中 → **空转线程**浪费CPU资源
 - 进入等待状态(sleep/wait)，释放CPU
- 当有新数据或工具可用时
 - notify/notifyAll：我准备好了新数据，或者我暂时不用这个工具了
 - JVM**唤醒**处于等待中的相关线程
- 如果有多个线程在等待呢？
 - JVM的唤醒选择具有**不确定性**

让**共享对象**来发出调度指令是个合理的选择！

如果共享对象不理朝政呢

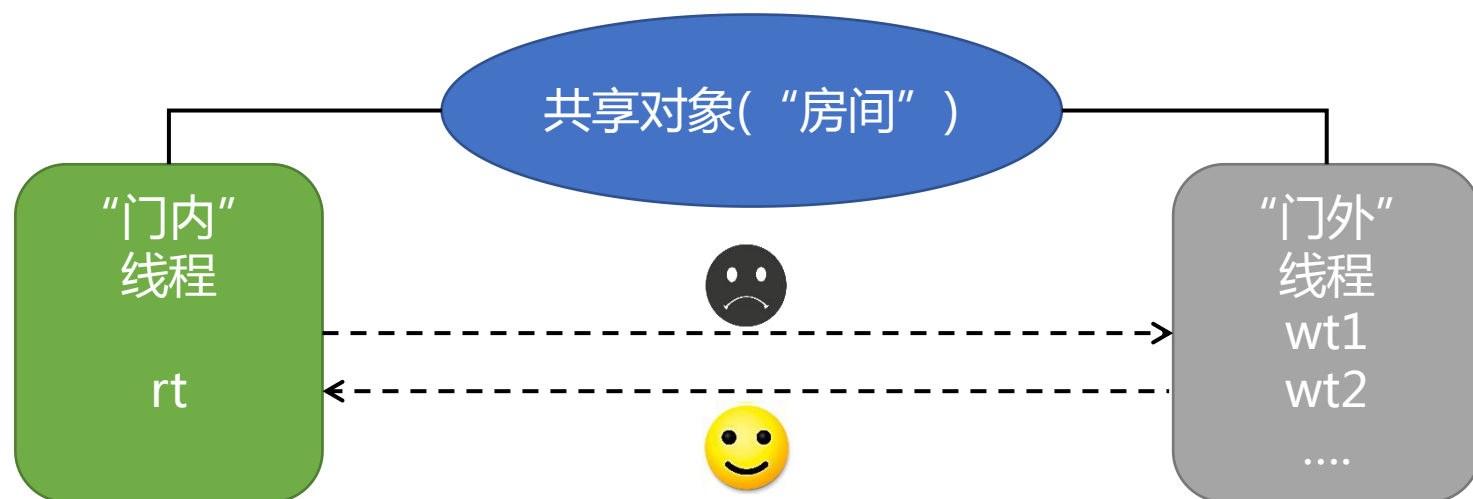
- 经典场景：多个线程对共享对象进行读写
 - 如果不加控制，读和写的次序是不确定的
 - 出现的现象称为**数据竞争**、读写**不一致**

Thread t1	<pre>public class BankAccount { private int balance; public void deposit(int amount) { balance += amount; } }</pre>	Thread t2
<pre>a.deposit(amount=50) : x = this.getBalance() x += amount; //2 this.setBalance(x) //3</pre>		<pre>deposit(amount=40) : // deposit 2 x = this.getBalance() //4 x += amount; //5 this.setBalance(x) //6</pre>

如果a和b是相同对象，且初始余额为0，1,4,2,5,3,6的执行顺序会导致什么结果？

共享对象应如何控制？

- **互斥**是根本：任何时刻只允许一个线程访问我的**数据或行为**
 - `synchronized(obj) {...}`：任意时刻只允许一个线程访问obj
 - `synchronized method(...){...}` 任意时刻只允许一个线程调用方法method
- `synchronized`修饰一个语句块：同步控制块（临界区）
 - 告诉JVM要对这个语句块进行线程互斥控制
 - 语句块执行结束前通过**notify/notifyAll**来通知JVM



如果**同步控制块**中
代码需要循环等待
新数据或工具可用
怎么办？

synchronized是如何发挥作用的

- 每个synchronized block都关联到一个对象(monitor)
 - JVM确保每个对象只有一个lock
 - JVM确保拿到monitor.lock的线程才能进入执行
- 场景1：作用于方法声明的同步块
 - 针对static方法：monitor实际是****.class
 - 效果：one thread per class
 - 针对非static方法：monitor实际是this
 - 效果：one thread per object/instance
- 场景2：作用于方法中局部代码的同步块
 - 取决于所选择的monitor对象
 - 效果：one thread per monitor

```
public class SynBlock {  
    public synchronized void m1() {  
        ...  
        m2();  
    }  
    public synchronized void m2() {  
        ...  
    }  
}
```

m1方法中对m2的调用是否会阻塞？

在何处通知JVM进行调度？

- 在线程所管理的执行流中的任何地方都有效果！
- 我们推荐简单实用的规则
 - 线程对象的代码中
 - 共享对象的代码中
- 线程对象代码
 - 一般使用sleep，wait来进行轮询
 - 或者使用通过join来等待其他线程执行完毕
- 共享对象代码
 - 建立synchronized临界区
 - 退出临界区前notify/notifyAll其他线程

单线程程序与多线程程序的对比

对比条目	单线程程序	多线程程序
流程数目	只有一个执行流	一到多个执行流
流程控制	方法调用、分支控制	方法调用、分支控制； 创建、启动、等待、唤醒
调试侵入	不影响执行结果	影响执行结果
对象交互	同步方式	同步方式、异步方式、并发方式

非线程间共享对象	线程间共享对象
遵循对象构造和引用基本规则	遵循对象构造和引用基本规则
相互间可以访问，无需额外控制	相互间可以访问，需要互斥控制
可访问线程间共享对象，遵循互斥规则	不可以 访问非线程间共享对象

同步和异步的对象交互

- 对象存在于堆中，相互独立，内在具有**并发性**
 - 对象调用方式的交互具有**同步特性**，需等待被调用方法返回才能继续向下执行
- 异步方式
 - 方法调用不必等待执行结束就立即返回：t.start()
- 基于共享对象的异步交互
 - 使用**信使对象**（共享对象）通知对方自己“做了什么”或者“状态发生了改变”
 - 例：下载对象通知界面对象下载进度的变化（**共享对象**）
- 彼此并发进行业务处理，通过**传送带**来交换数据
- 例：质量检测机器人持续检查合格品传送带上的货品，一旦发现残次品，拿出放入不合格品传送带

线程的基本设计框架

- 实现一个独立和完整的算法/功能
 - run方法
- 通过构造器或专门的方法获得与其他线程共享的对象
 - 数据交互窗口：线程间共享对象
 - 实现了Runnable接口的对象本身也可以作为共享对象
- 创建和使用专属对象
 - 仅供自己这个线程使用：非线程间共享对象
 - 这些对象之间仍然可以相互调用方法

线程的基本设计框架

- 通过共享对象与其他线程交互
 - 通过锁来确保任何时候只能有一个线程在共享对象 “房间” 内工作
 - 基于obj的语句段级同步控制: `synchronized(obj){}`
 - 基于this的方法级同步控制: `synchronized method{}`
 - 完成工作即退出房间
 - 交出锁(自动)
 - **通知**其他在等待进入共享对象 “房间” 工作的线程
 - 继续 “自己家里” 的处理工作

直接调用run与线程调度执行run的区别？

- 程序中有一个类AThread实现了Runnable接口，并在main方法一开始就创建了如下两个对象
 - AThread at = new AThread();
 - Thread t1 = new Thread(at);
- 对比如下三种main方法实现，分析t1线程启动执行到run时的counter值

```
public class AThread implements Runnable{  
    private int counter=0;  
    public void run(){  
        counter++;  
        System.out.println("current thread:" +  
            Thread.currentThread().getName() +  
            "\tcounter:" + counter);  
    }  
}
```

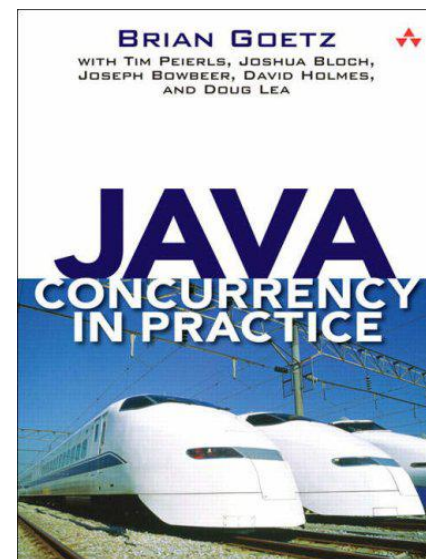
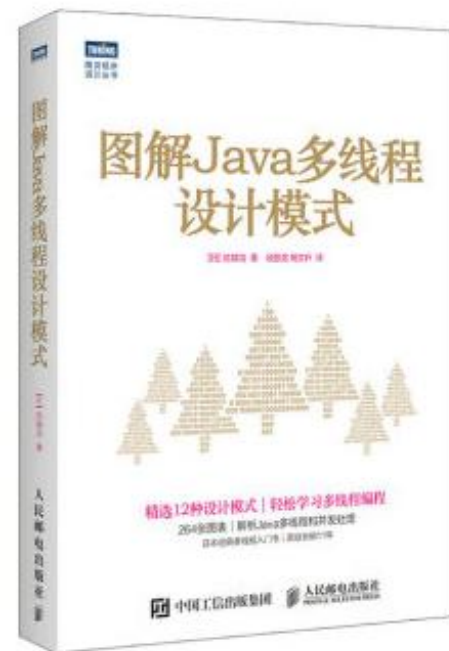
```
...main(String[] args){  
    ...  
    t1.start();  
}
```

```
...main(String[] args){  
    ...  
    t1.start();  
    t1.run();  
}
```

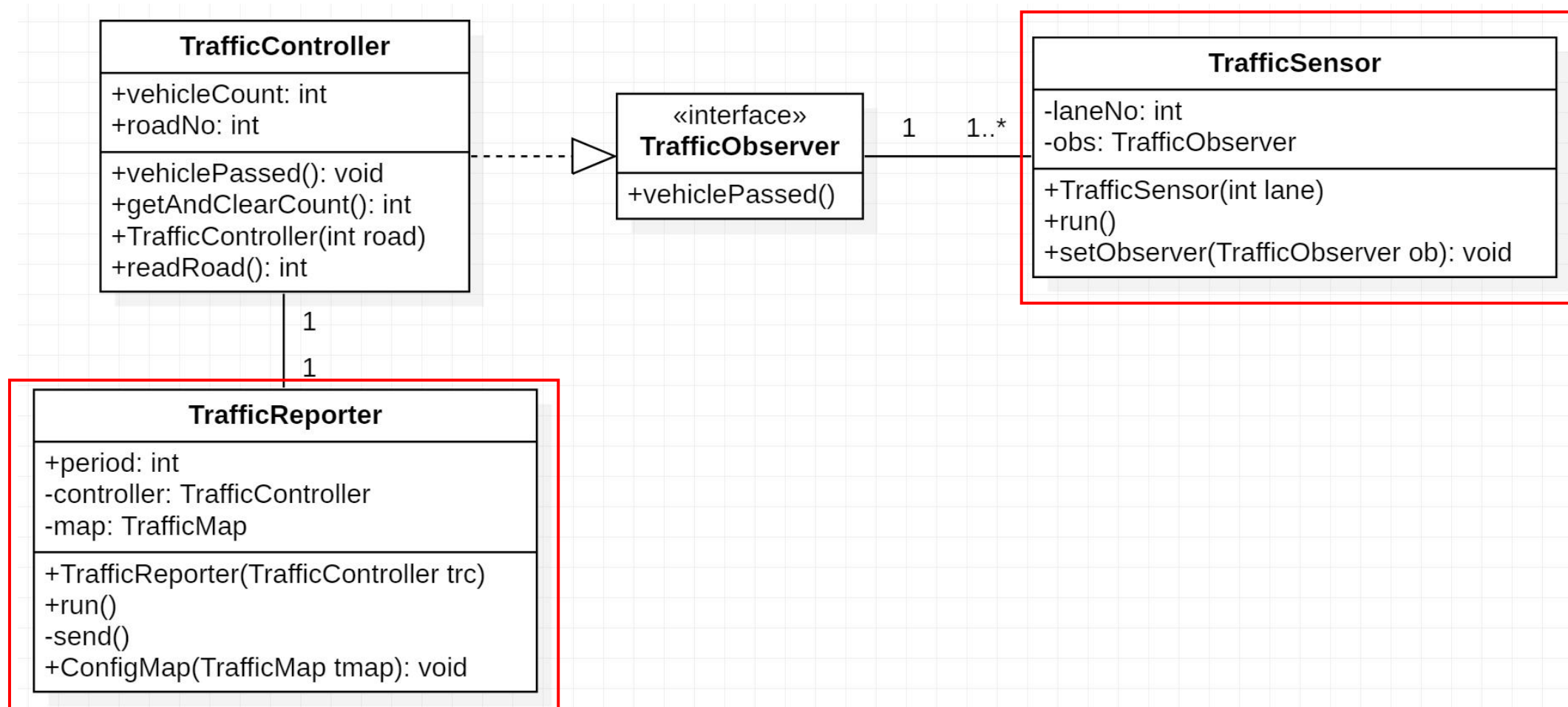
```
...main(String[] args){  
    ...  
    at.run();  
    t1.start();  
}
```

多线程交互模式：线程互斥访问

- 多个线程共享访问一个资源，限制任何时候只能有一个线程访问
 - 基本做法：把共享对象的方法进行synchronized保护
- 案例问题
 - 智能交通系统需要实时统计车流，从而及时对车辆出行进行调度和控制
 - 假设有一种摄像头可以自动识别经过的车辆
 - 每个车道都安装有这样的摄像头
 - 系统能够按照一定的周期把通过的车辆统计数据上传



多线程交互模式：线程互斥访问



多线程交互模式：线程互斥访问

- 所有的车道传感器都会向controller报告检测的车辆通过信息
 - vehiclePassed必须进行同步保护，确保不会发生数据错误
- 车道传感器看不到getAndClearCount方法，reporter周期性进行访问，因此没必要保护？

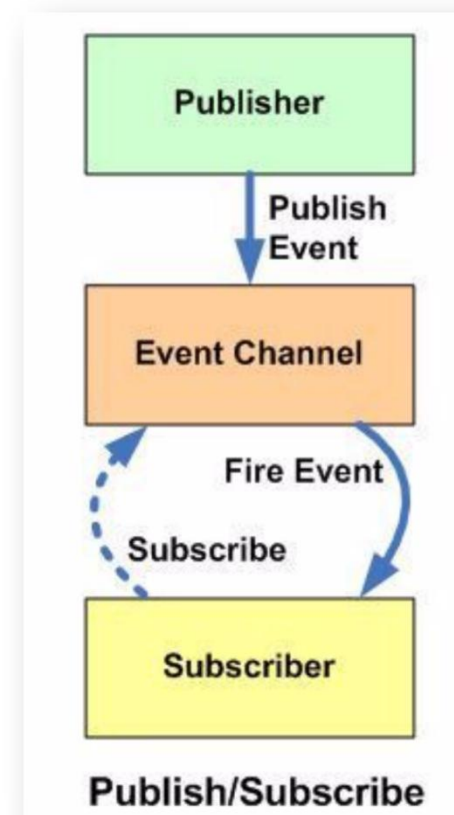
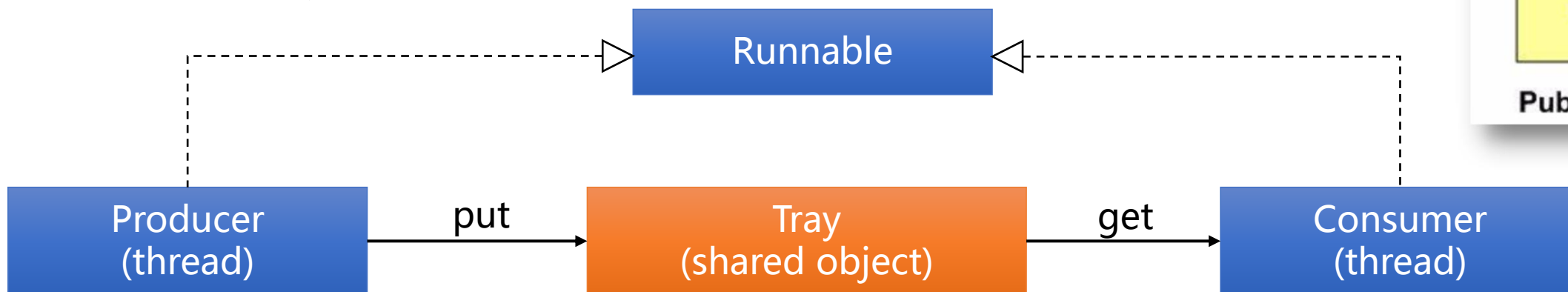
```
public class TrafficController {  
    private int vehicleCount;  
    private int roadNo;  
    public TrafficController(int road){  
        roadNo = road;  
        vehicleCount = 0;  
    }  
    public synchronized void vehiclePassed() {  
        //all sensors will call  
        vehicleCount ++;  
    }  
    public synchronized int getAndClearCount() {  
        //only the reporter will call  
        int cur_count = vehicleCount;  
        vehicleCount = 0;  
        return cur_count;  
    }  
}
```

多线程交互模式：线程互斥访问

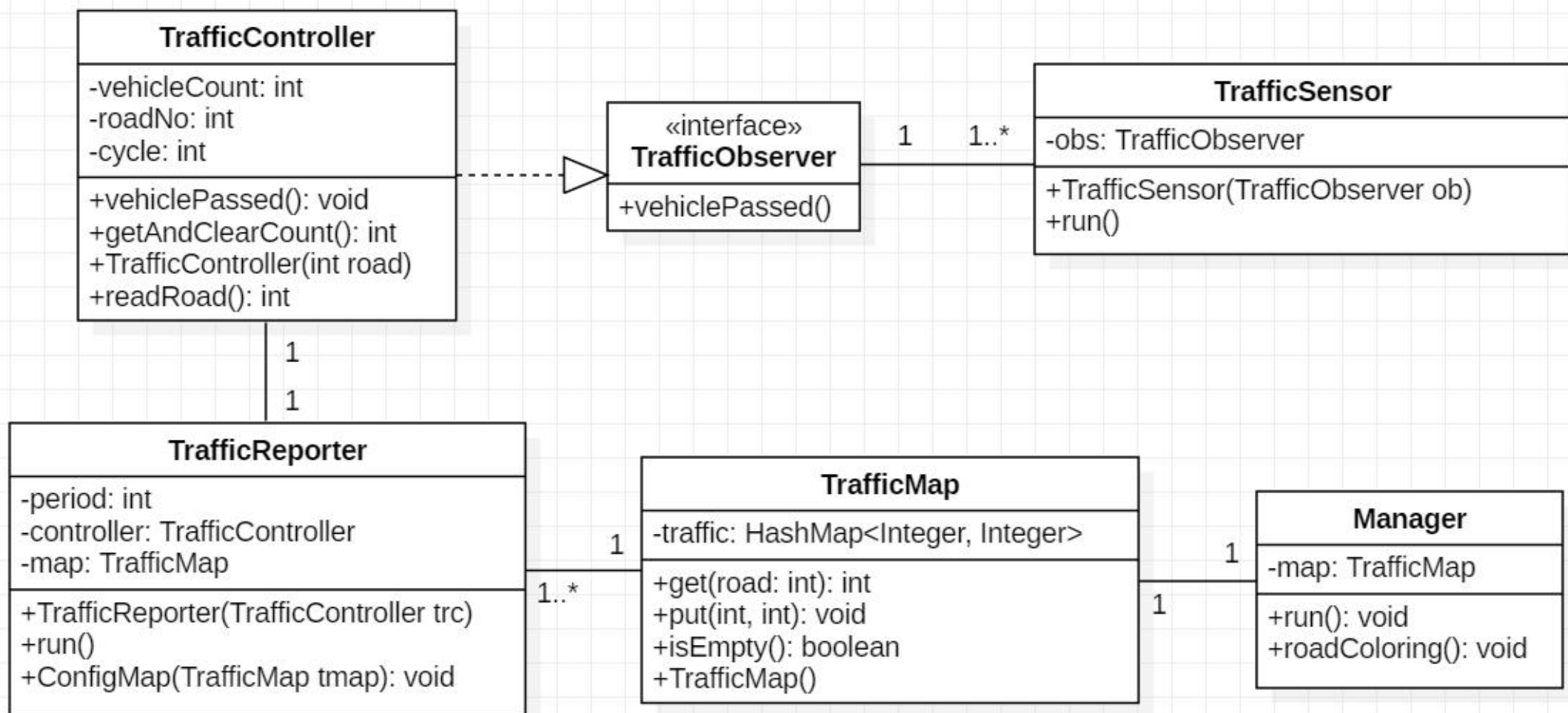
- 进一步的问题拓展
 - 每个路口都部署相应的局部系统，包括若干sensor，一个controller对象和一个reporter对象
 - 一个城市可能会有上万个reporter线程在并发报告车流信息
 - 上层的主线程会不断根据报告上来的车流信息，按照地图中道路之间的连接关系来评估道路的拥堵状况
- 如何扩展当前的设计？
 - 多个reporter不断报告新数据：需要一个缓冲区来进行管理
 - manager线程需要持续工作来消耗缓冲区的数据
 - ➔生产者-消费者模式

线程交互模式：生产者消费者

- 经典问题：生产者和消费者
 - 生产者向托盘对象存入生产的货物 //synchronized method
 - 消费者从托盘里取走相应的货物 //synchronized method
 - 货物放置控制 //依赖于托盘状态
 - 货物提取控制 //依赖于托盘状态
 - 托盘既可以是单一对象，也可以是容器对象
- 又可称为发布订阅模式



线程交互模式：生产者消费者



线程交互模式-生产者

- TrafficReporter按照设定的周期读取局部的车流计数，形成“产品”放入“传送带”中
 - 产品：<road, count>
 - 传送带：map
- 该传送带自我管理容量，生产者无需担心是否已满

```
public class TrafficReporter implements Runnable{
    private TrafficController controller;
    private TrafficMap map;
    private int period;
    public TrafficReporter(TrafficController trc){
        period = 10;
        controller = trc;
        map = null;
    }
    public void ConfigMap(TrafficMap tmap) {
        map = tmap;
    }
    public void run() {
        try {
            while(true) {
                Thread.sleep(period*10);
                int count = controller.getAndClearCount();
                int road = controller.readRoad();
                map.put(road, count);
            }
        } catch (InterruptedException e) {}
    }
}
```

线程交互模式-消费者

- Manager持续不断的扫描“传送带”，只要有新的“物品”，就取走
 - 物品：<road, count>
 - 传送带：map
- 传送带确保按照“商品编号”进行打包，使得消费者一次提取就可以把相应商品编号的物品全部提走

```
public class Manager implements Runnable{
    private TrafficMap map;
    private int roadMAX = 1000;
    Manager(TrafficMap tmap){
        map = tmap;
    }
    public void run() {
        int count;
        while (true) {
            for(int r = 0; r < roadMAX; r++) {
                count = map.get(r);
                //roadColoring(r, count);
            }
        }
    }
}
```

线程交互模式-可伸缩的传送带

- 使用可伸缩的容器来管理产品及其传递
 - HashMap
- 自动根据产品编码来进行打包合并
 - 提高消费者的工作效率
- 注意确保所有访问者的公平性
 - notifyAll

```
public class TrafficMap {
    private HashMap<Integer, Integer> traffic;
    TrafficMap(){traffic = new HashMap<Integer, Integer>();}
    public synchronized void put(int road, int vcount) {
        Integer r = new Integer(road);
        if(traffic.containsKey(r))
            traffic.put(r, new Integer
                (traffic.get(r).intValue()+vcount));
        else traffic.put(r, new Integer(vcount));
        notifyAll();
    }
    public synchronized int get(int road) {
        Integer r = new Integer (road);
        while (!traffic.containsKey(r)) {
            try { wait();}catch (InterruptedException e) {}
        }
        t = traffic.get(r); traffic.remove(r);notifyAll();
        return t.intValue();
    }
}
```

多线程交互模式：容量固定的传送带

```
public class NonScalableTray {  
    private int value;    private boolean full = false;  
    public synchronized int get() {  
        while (full == false) {  
            try { wait(); } catch (InterruptedException e) { }  
            full = false;  
            notifyAll();  
            return value;  
        }  
    }  
    public synchronized void put(int v) {  
        while (full == true) {  
            try { wait(); } catch (InterruptedException e) { }  
            full = true;  
            value = v;  
            notifyAll();  
        }  
    }  
}
```


多线程交互模式：生产者消费者

- 请同学们自行补充和完成演示
 - 实现TrafficSensor类
 - 每个road对应若干TrafficSensor线程实例
 - 创建若干TrafficReporter线程实例
 - 每个road对应一个reporter线程
 - 创建一个Manager线程实例
 - 通过TrafficMap和所有的TrafficReporter线程进行交互
- 可以构造生产者-消费者链，实现pipeline结构
 - Sensor--Reporter--Manager
- 生产者与消费者之间可以是多对多的关系
 - 生产多种物品
 - 消费多种物品

```
Problems Javadoc Declaration Console Progress
<terminated> CMain [Java Application] C:\Program Files\Java\jre1.8.0_241\bin
vechile passed:7 in road: 0 for the cycle: 1
vechile passed:7 in road: 1 for the cycle: 1
vechile passed:10 in road: 2 for the cycle: 1
record added into map: (road 0, vechile passed count 7)
record added into map: (road 2, vechile passed count 10)
record added into map: (road 1, vechile passed count 7)
record fetched from map: (road 0, vechile passed count 7)
mgr received: (road: 0, vechile passed count: 7)
record fetched from map: (road 1, vechile passed count 7)
mgr received: (road: 1, vechile passed count: 7)
record fetched from map: (road 2, vechile passed count 10)
mgr received: (road: 2, vechile passed count: 10)
vechile passed:8 in road: 0 for the cycle: 2
record added into map: (road 0, vechile passed count 8)
vechile passed:9 in road: 2 for the cycle: 2
record fetched from map: (road 0, vechile passed count 8)
mgr received: (road: 0, vechile passed count: 8)
record added into map: (road 2, vechile passed count 9)
vechile passed:7 in road: 1 for the cycle: 2
record added into map: (road 1, vechile passed count 7)
record fetched from map: (road 1, vechile passed count 7)
mgr received: (road: 1, vechile passed count: 7)
record fetched from map: (road 2, vechile passed count 9)
mgr received: (road: 2, vechile passed count: 9)
vechile passed:9 in road: 2 for the cycle: 3
record added into map: (road 2, vechile passed count 9)
vechile passed:7 in road: 0 for the cycle: 3
vechile passed:9 in road: 1 for the cycle: 3
record added into map: (road 0, vechile passed count 7)
record added into map: (road 1, vechile passed count 9)
record fetched from map: (road 0, vechile passed count 7)
```

常见的临界区设置错误

- **synchronized**只能在同步范围内发挥作用

```
class CustomerUpdater {  
    CustomerUpdater(CustomerDb cdb) {...}  
    public void run(){  
        ArrayList<Customer> customers =  
            cdb.getCustomers();  
  
        ..  
        customers.add(customer);  
    }  
}
```

```
class CustomerDb {  
    private ArrayList<Customer> customers  
        = new ArrayList();  
    public synchronized ArrayList<Customer>  
        getCustomers(){  
        return customers;  
    }  
}
```

synchronized定义了代码层次的临界区，
对临界区外的共享访问**不产生作用**！

常见的临界区设置错误

- 所有对共享对象的访问都需要进行同步控制，不能遗漏

```
class CustomerDb2 {  
    private final ArrayList<Customer> customers = new ArrayList();  
    public addCustomer(Customer c){  
        synchronized (customers){  
            customers.add(c);  
        }  
    }  
    public removerCustomer(Customer c){  
        customers.remove(c);  
    }  
}
```

既然customers是共享对象，**任意时刻**都可能有**多个线程**在访问它！

常见的临界区设置错误

- 同步控制施加在上层对象，对下层的共享对象不产生作用

```
class CustomerDb3 {  
    private final List<Customer> customers = new ArrayList();  
    public double countOrderValue (Customer c){  
        ArrayList<Order> orders;  
        synchronized (c){  
            orders = c.getOrders();  
        }  
        double total = 0.0;  
        for(Order o : orders)  
            total += o.getValue();  
        return total;  
    }  
}
```

c是**共享对象**，c所管理的orders
也是**共享对象**！

作业

- 多线程电梯系统
 - 一部电梯，各项参数静态不可变（运行速度、开门、关门、限乘人数、楼层数、初始位置）
 - 提供输入和输出处理包，**聚焦于线程交互控制和电梯调度**
- 输入输出定义
 - 输入：[时间戳]id-FROM-A-x-to-B-y：乘客从A楼座的x层到B楼座的y层
 - 输出：[时间戳]电梯ID-动作
 - 电梯动作：开门、关门、到达楼层
 - 乘客动作：进入、离开电梯
- 调度策略
 - 自行定义，但必须确保所有有效请求得到了响应
 - 你的调度应至少和ALS策略的调度性能相当

设计建议

- 设计多少个线程？输入？电梯？调度器？
 - 1、请求输入装置和电梯是线程，共享调度器对象，向调度器发布请求和读取请求
 - 调度器管理请求队列，外部不可知
 - 队列和楼层之间的关系是否需要更便捷的管理？
 - 2、请求输入装置、电梯和调度器是线程
 - 这三个线程之间的生产-消费关系是什么？共享对象是什么？
 - 调度器与请求队列分离，三个线程都依赖于请求队列这个共享对象
- 如何调度电梯？
 - 候乘表类、调度策略类
 - 调度：分析候乘表类中列出的“任务”，确定下一个目标楼层，然后转化为一系列电梯运行动作