《面向对象设计与构造》Lec05-对象并发及其协同

北京航空航天大学计算机学院

OO课程组2024

吴际

提纲

- 单元主题分析
- Java程序的运行机制
- 多线程程序框架
- 线程状态与调度机制
- 线程行为的不确定性及其控制
- 几种线程交互模式
- 本周作业解析

单元主题分析

- 对象行为之间具有多种关系
  - 顺序执行关系：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？ ➔ 引入了不确定性

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

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

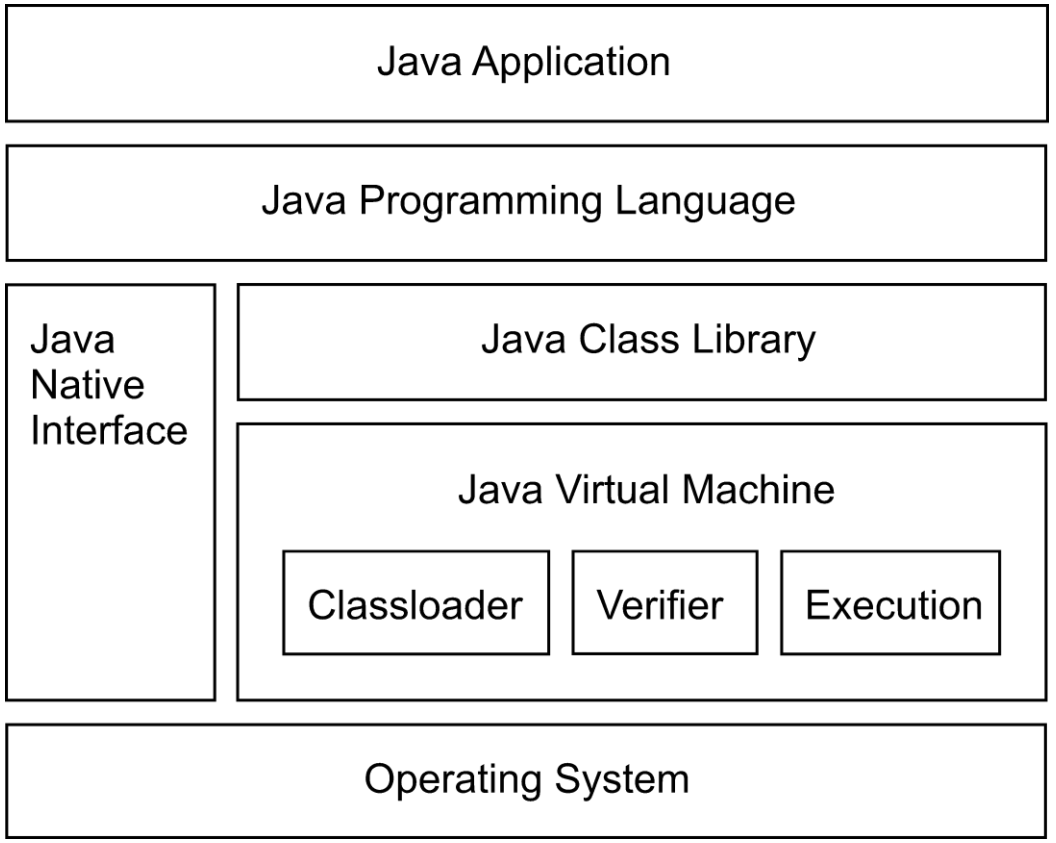
- Java程序的运行时系统原生支持了对象并发执行

Java程序的运行时系统

你编写的Java程序

Java编译器

本地程序调用接口



Java类库程序包

Java程序执行虚拟机

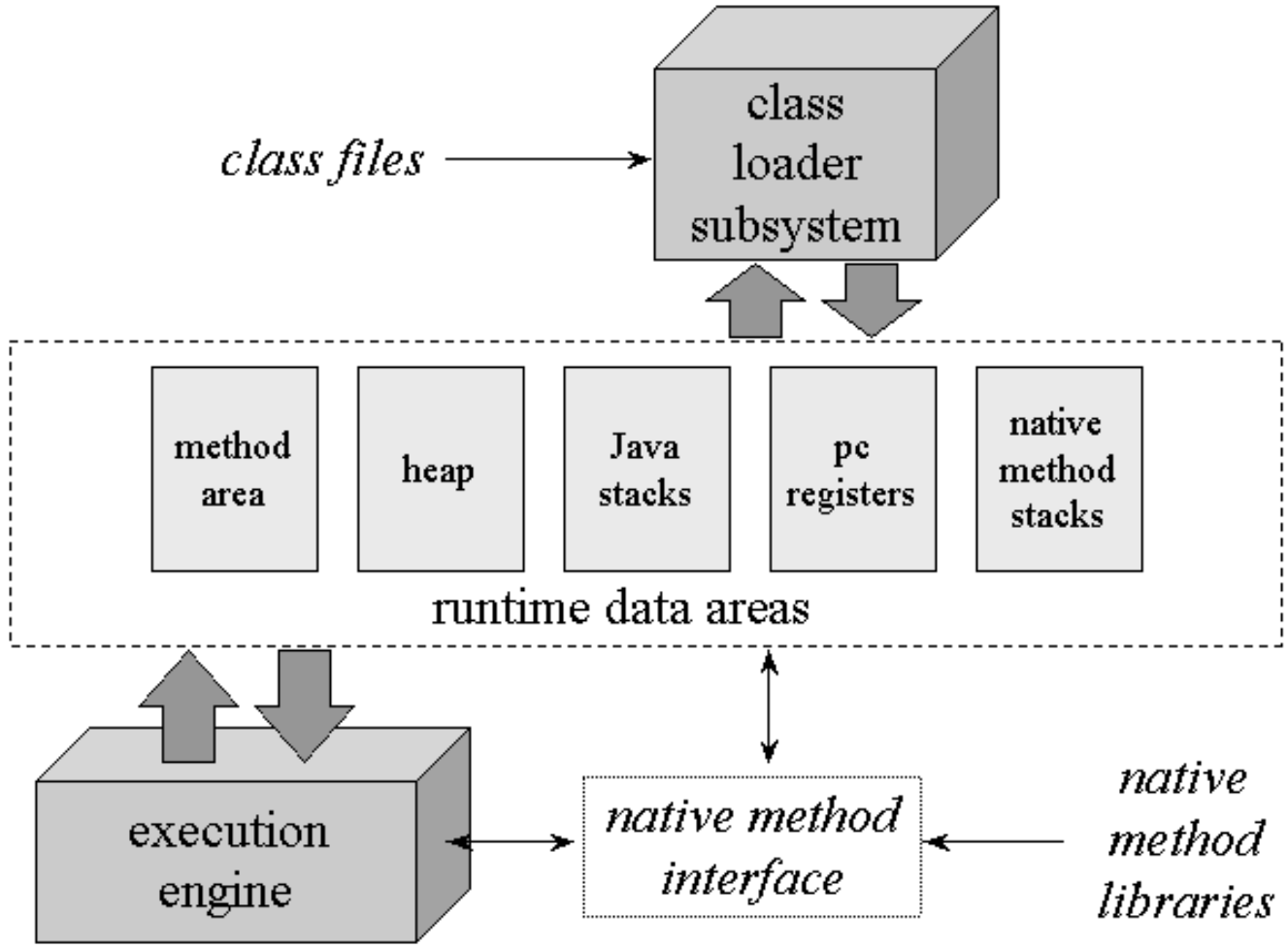
操作系统

Java程序在JVM中执行

待执行指令代码

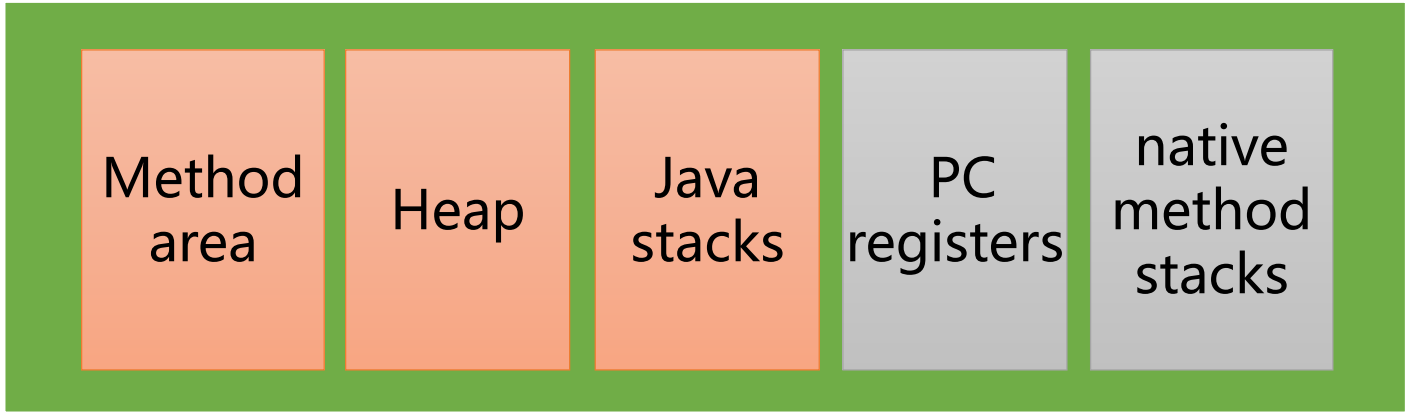
内存数据结构

指令执行引擎



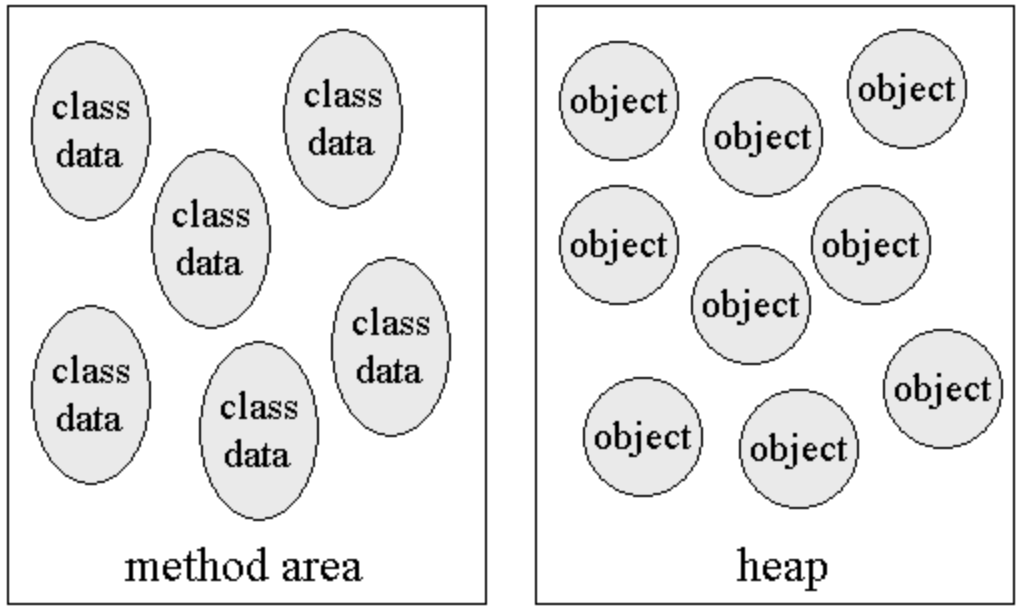
JVM内存区域划分

- Java程序运行时由一到多个**线程**实例组成
- 每个线程实例都有自己的栈(**局部栈**)
- 所有线程实例共享访问同一个堆(**全局堆**)

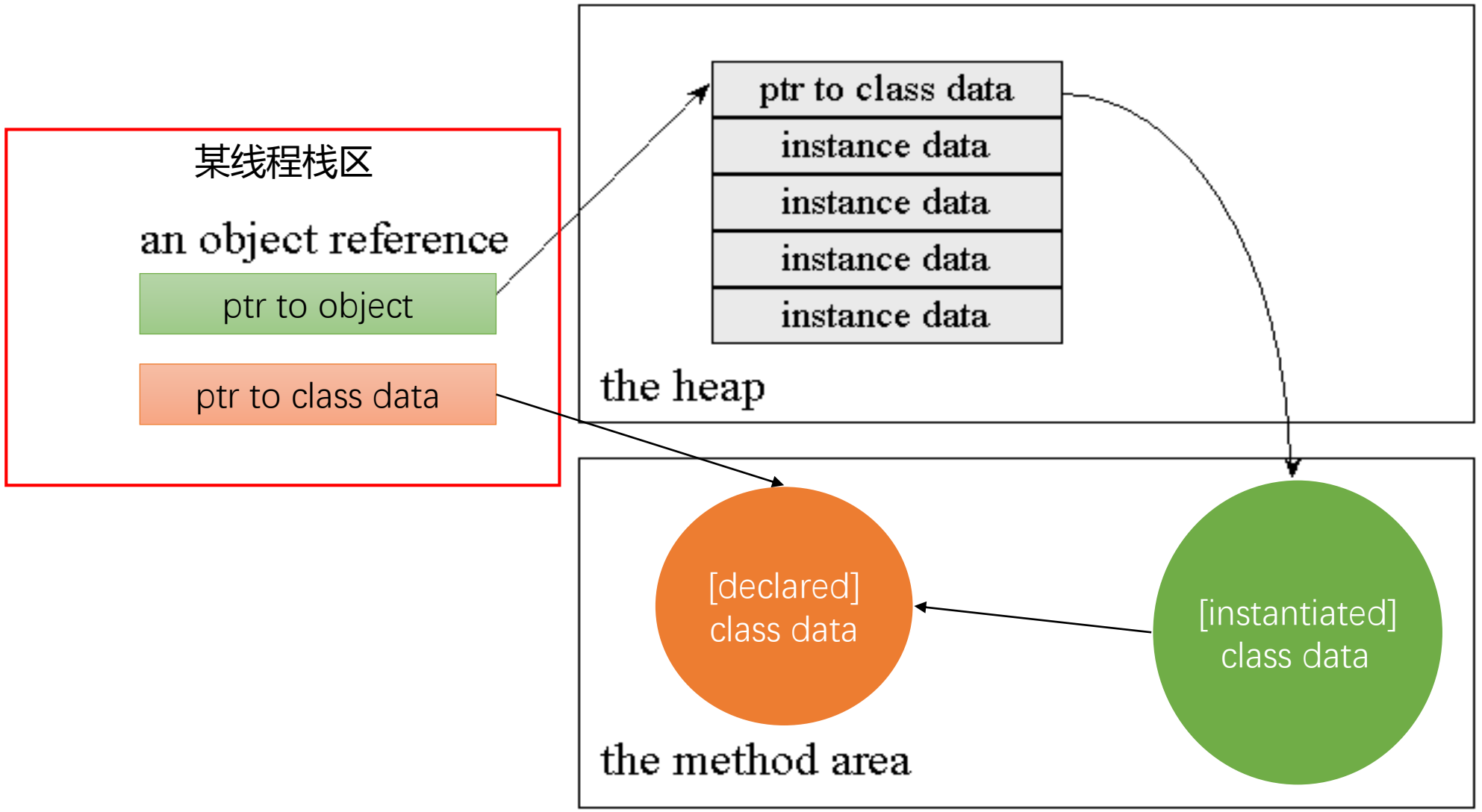


JVM内存区域划分

- 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访问
  - **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
  - 虚拟机中所有线程共享访问
  - 为垃圾回收提供支持
  - 程序执行过程中动态扩展和收缩

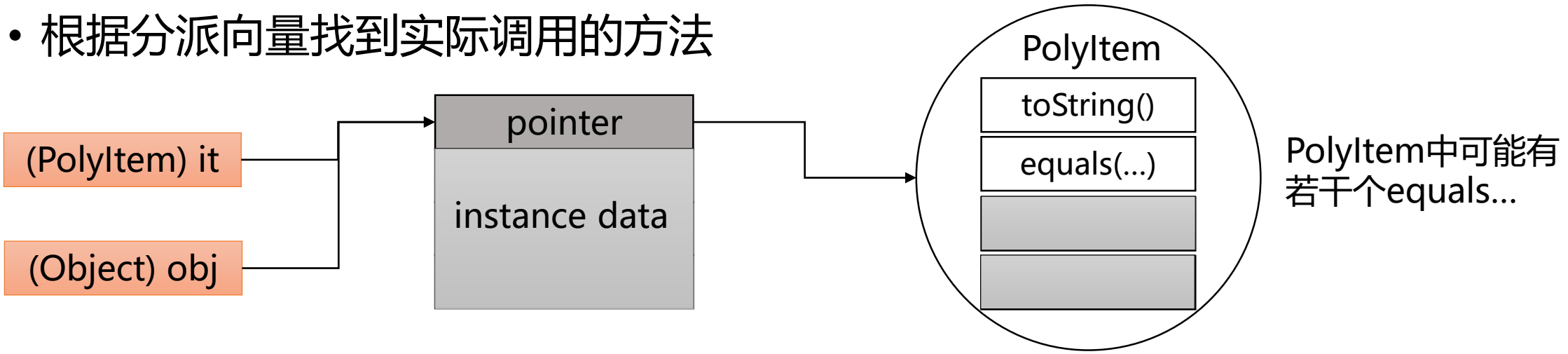


连接起来才能工作


某线程栈区
an object reference
ptr to object
ptr to class data
ptr to class data
instance data
instance data
instance data
instance data
the heap
[declared]
class data
[instantiated]
class data
the method area

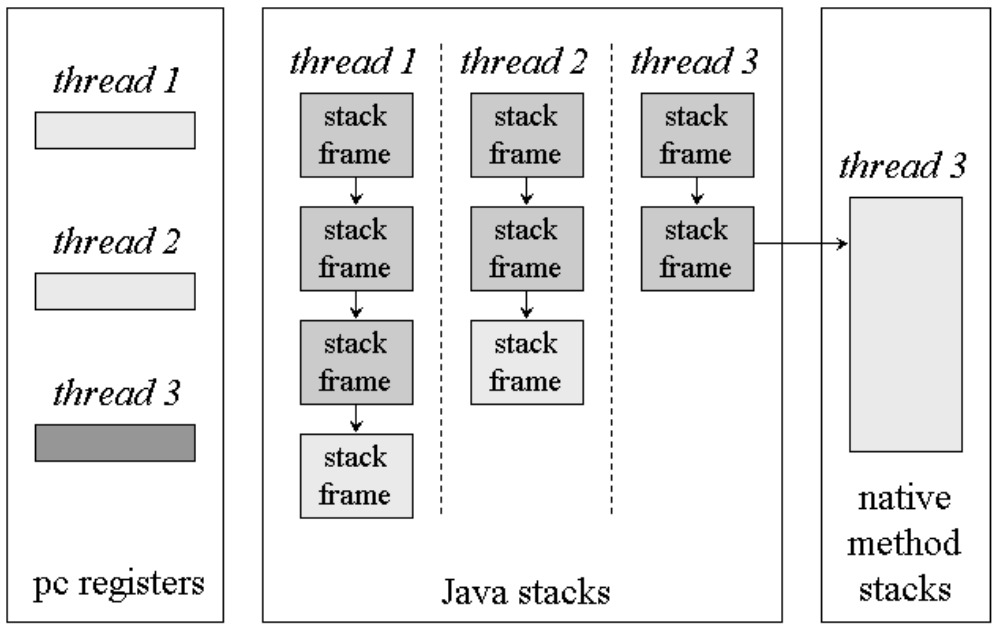
如何知道调用哪个方法?

- 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



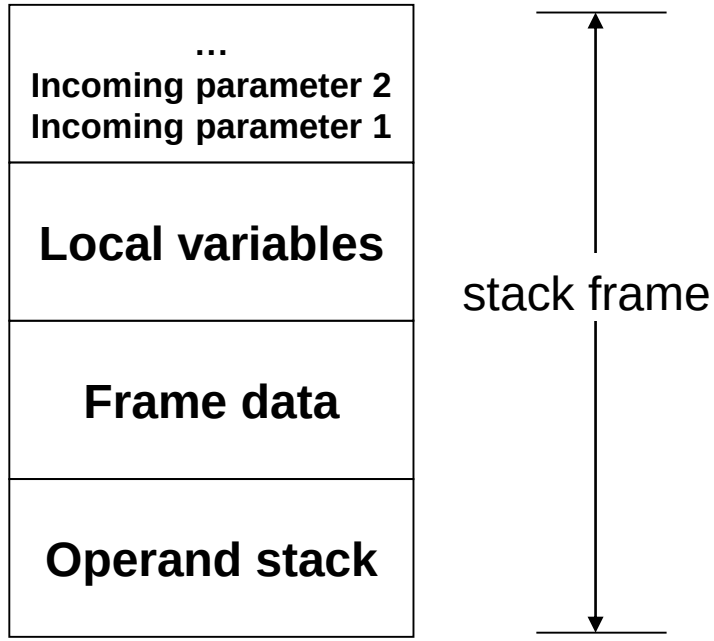
JVM栈的内存结构

- 栈内存区
  - 每个线程都拥有专属的栈内存，用以追踪方法的调用关系
  - 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



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程序中的对象引用(变量)存在何处?

Java程序运行时的内存状态变化

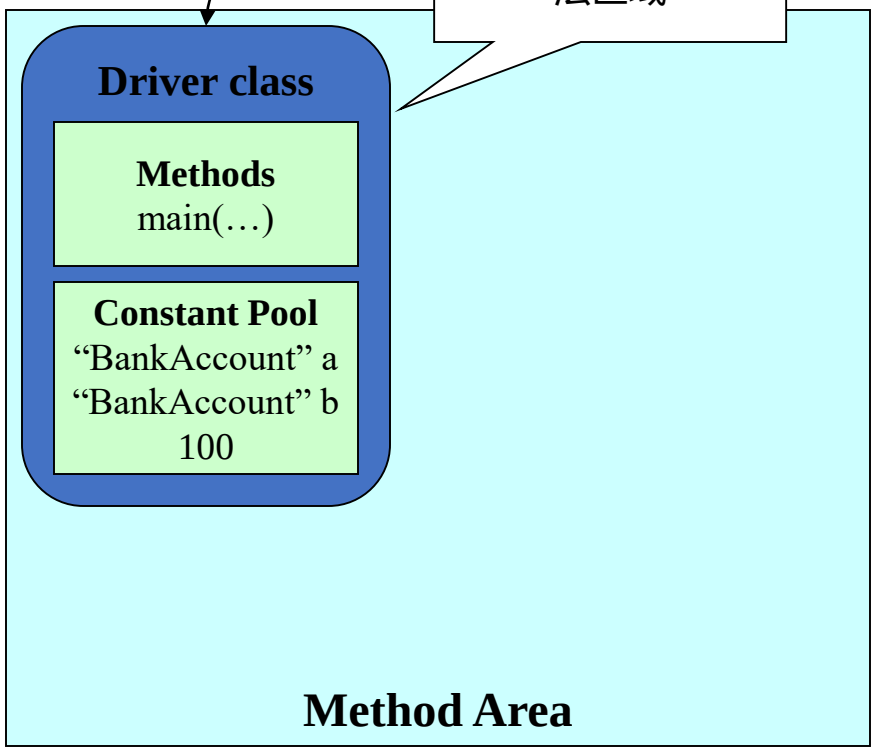
```
public class BankAccount {
    private double balance;
    private static int totalAccounts = 0;
    public BankAccount() {
        balance = 0;
        totalAccounts++;
    }
    public void deposit( double amount ) {
        balance += amount;
    }
}
```

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

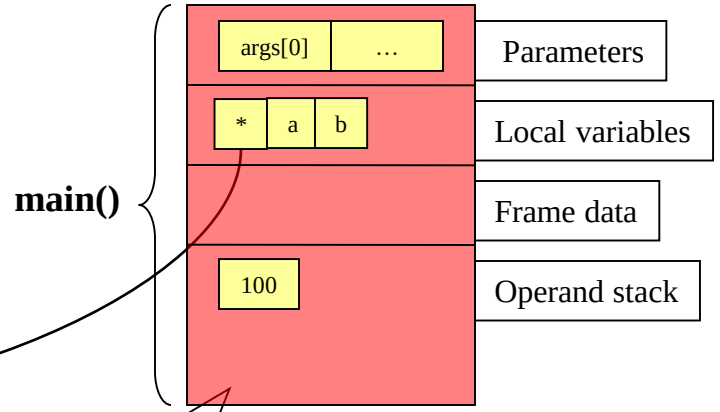
```
// In command prompt
java Driver

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

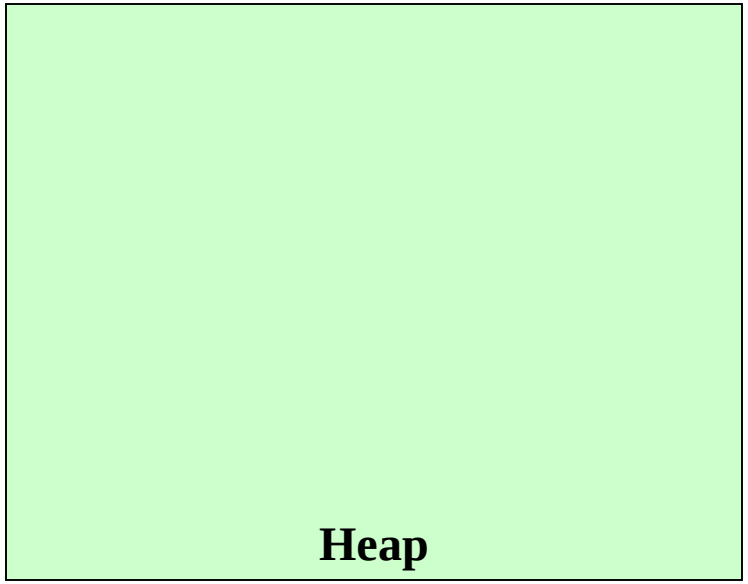
```
public class Driver {
    public static void main( String[] args ) {
        BankAccount a = new BankAccount();
        BankAccount b = new BankAccount();
        b.deposit( 100 );
    }
}
```



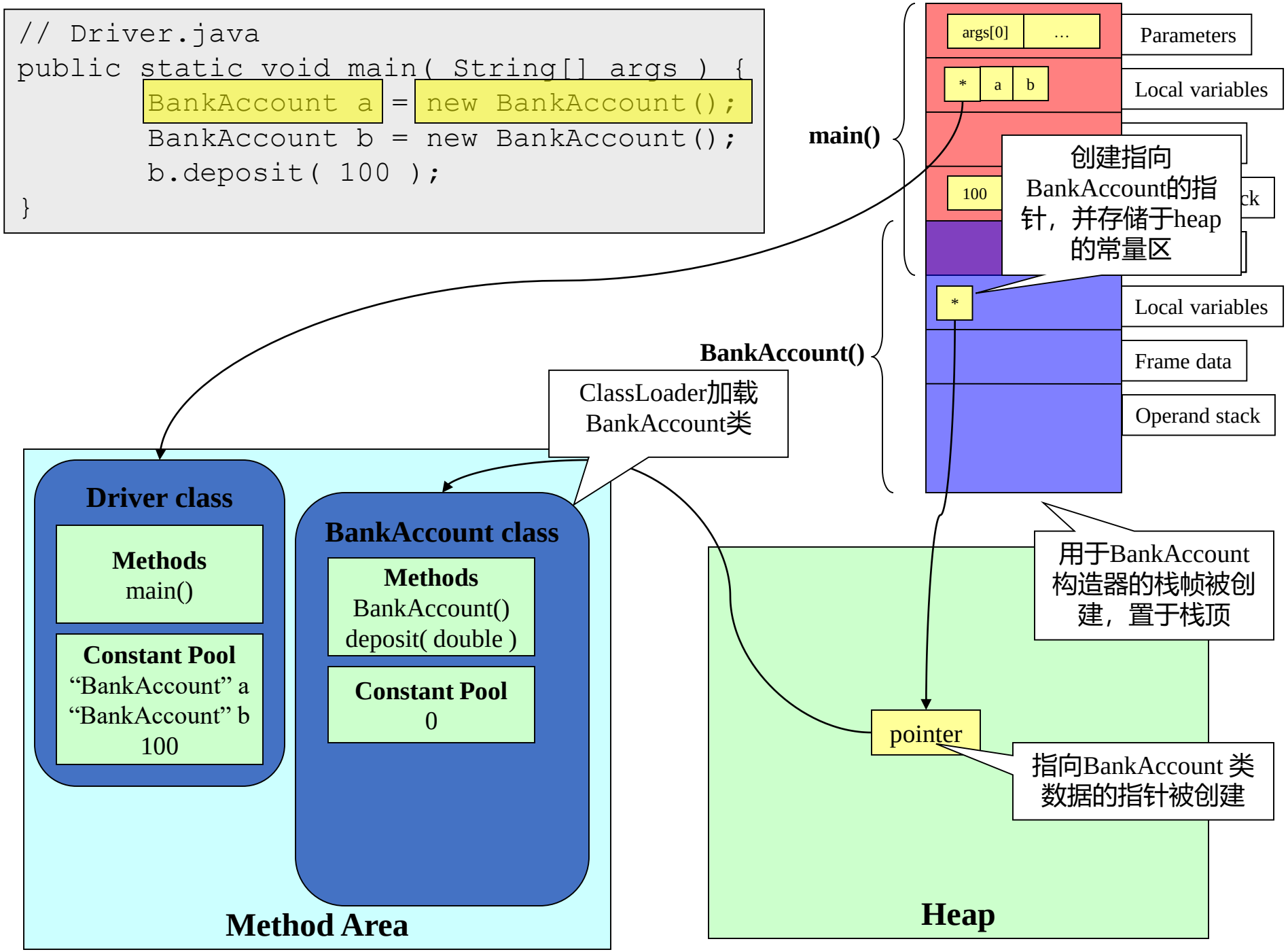
ClassLoader把Driver类加载到方法区域

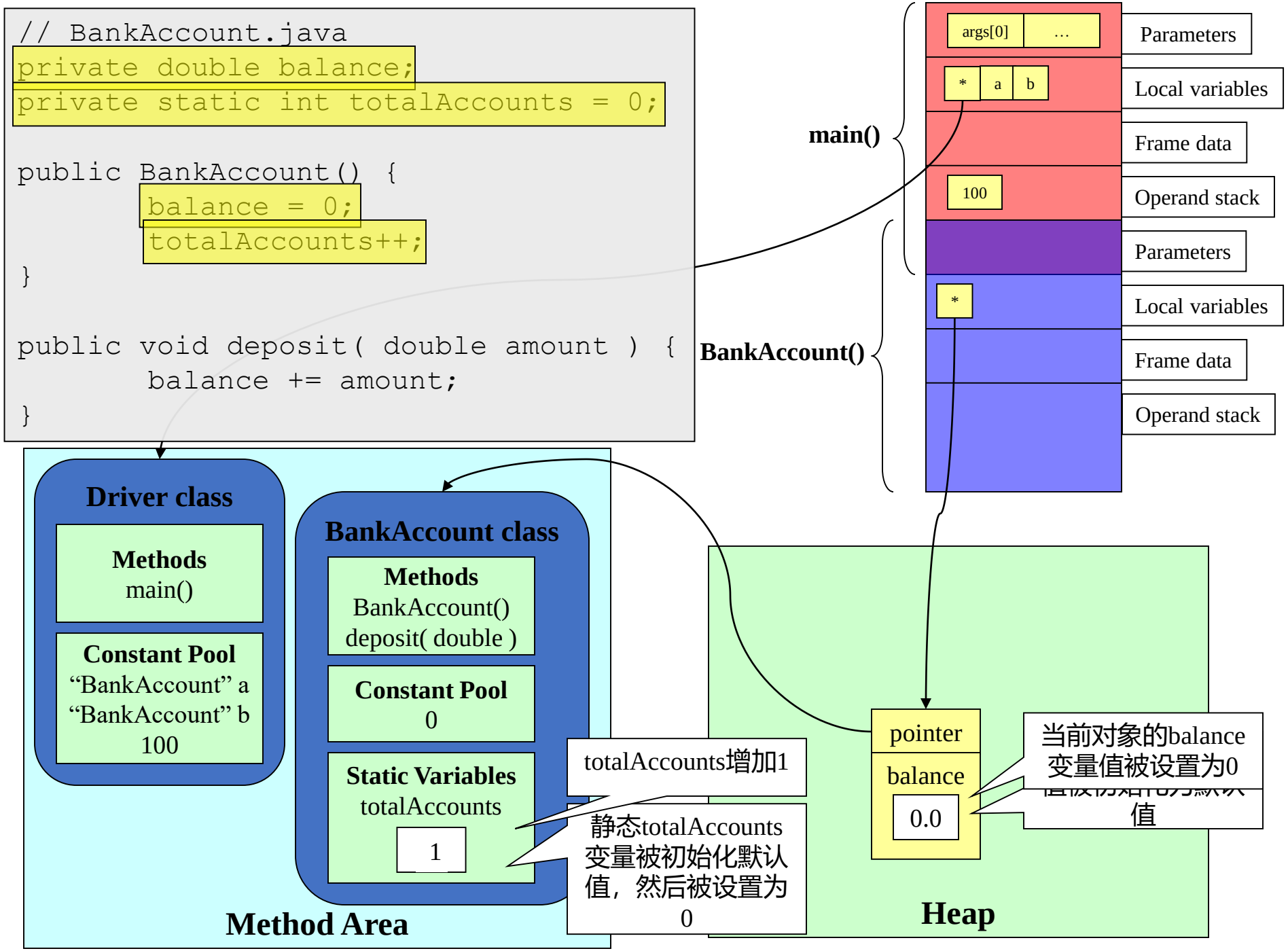


针对main()方法的栈帧被创建，并处于栈顶。*为对当前对象对应class的引用。

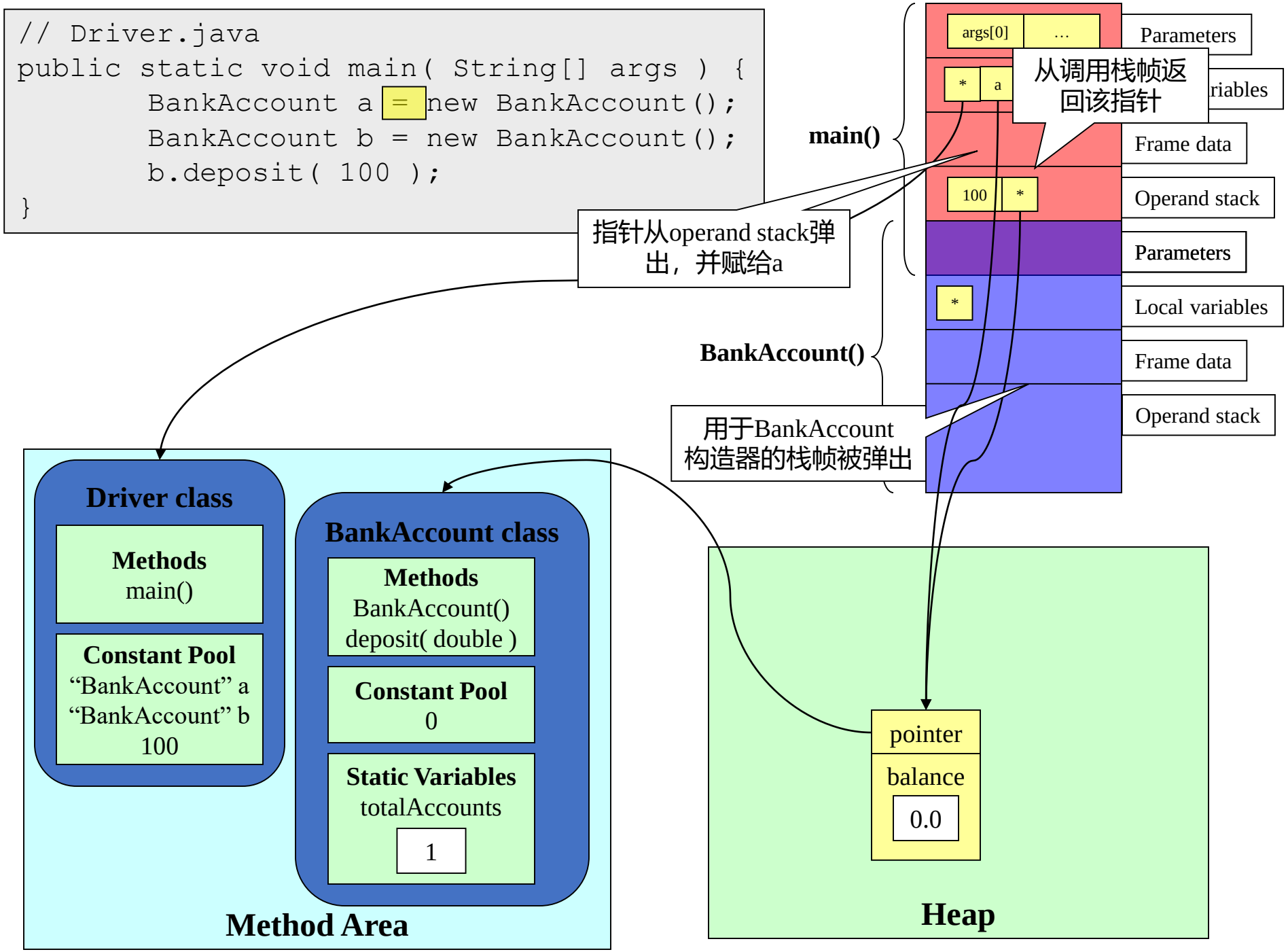


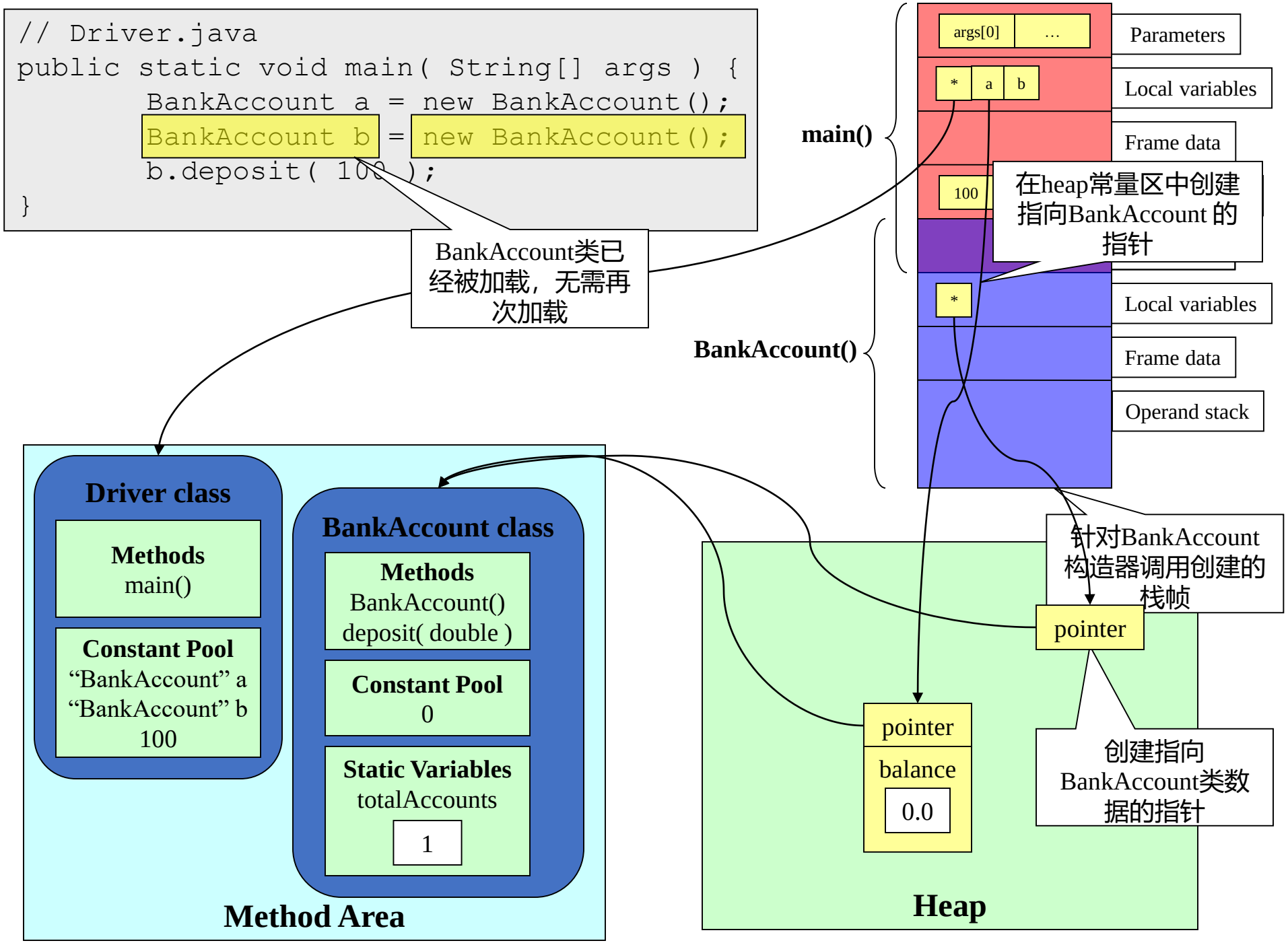
```
// Driver.java
public static void main( String[] args ) {
        BankAccount a = new BankAccount();
        BankAccount b = new BankAccount();
        b.deposit( 100 );
}
```

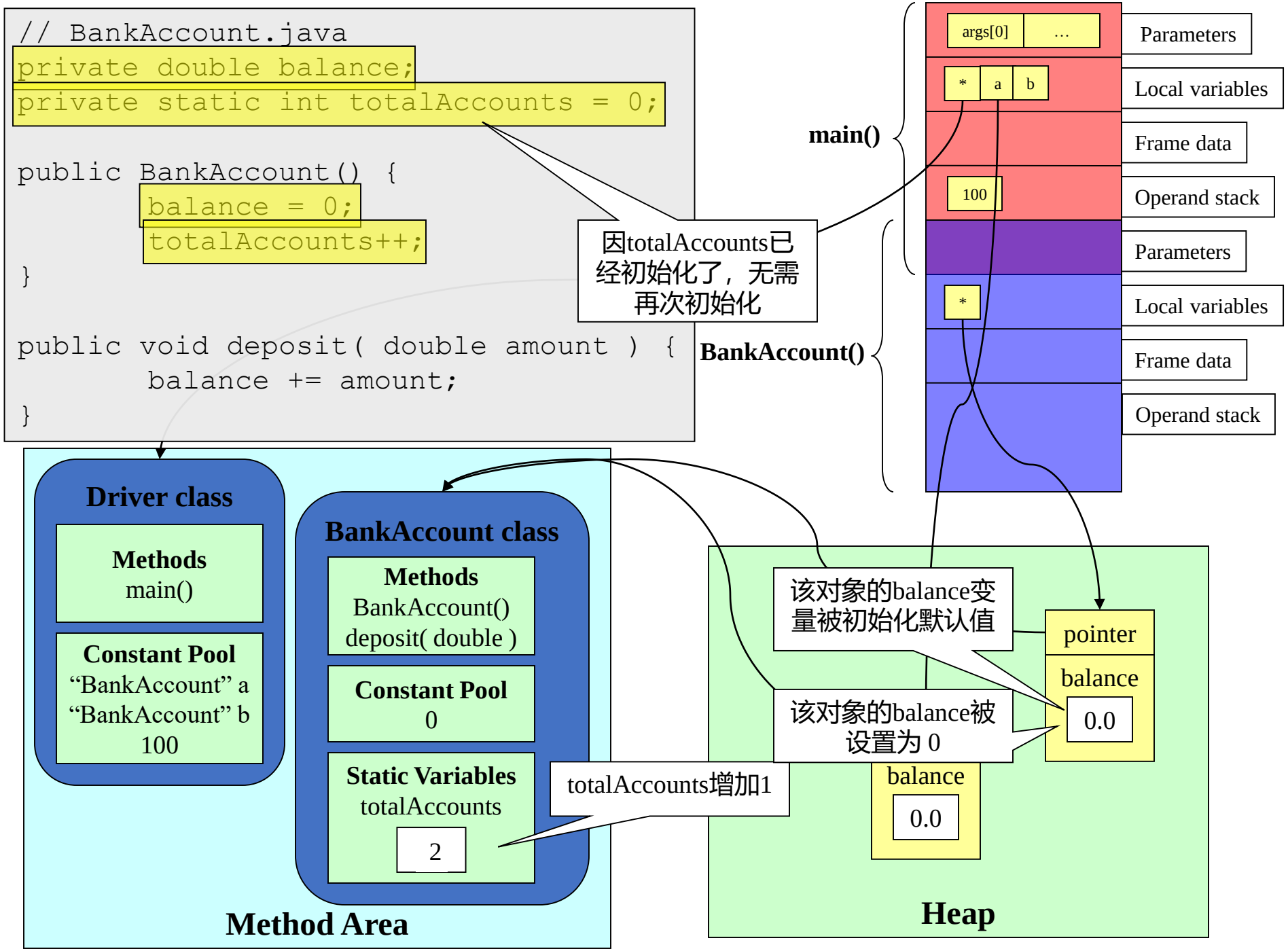



// BankAccount.java
private double balance;
private static int totalAccounts = 0;
public BankAccount() {
balance = 0;
totalAccounts++;
}
public void deposit(double amount) {
balance += amount;
}
main()
BankAccount()
args[0]
…
Parameters
*
a
b
Local variables
Frame data
100
Operand stack
Parameters
*
Local variables
Frame data
Operand stack
Driver class
Methods
main()
Constant Pool
“BankAccount” a
“BankAccount” b
100
BankAccount class
Methods
BankAccount()
deposit(double)
Constant Pool
0
Static Variables
totalAccounts
1
Method Area
totalAccounts增加1
静态totalAccounts变量被初始化默认值，然后被设置为0
pointer
balance
0.0
当前对象的balance变量值被设置为0
Heap

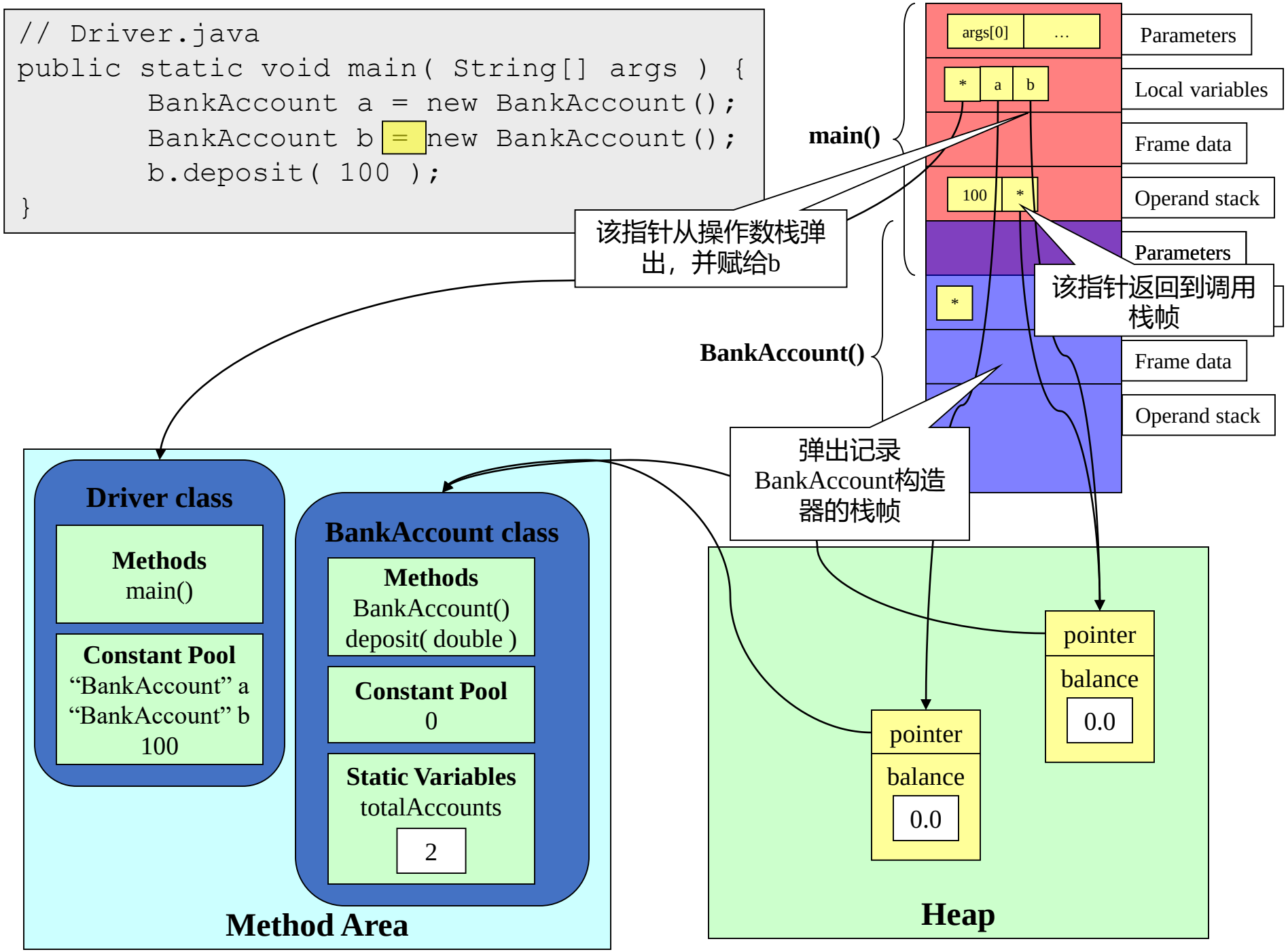
```
// Driver.java
public static void main( String[] args ) {
        BankAccount a = new BankAccount();
        BankAccount b = new BankAccount();
        b.deposit( 100 );
}
```



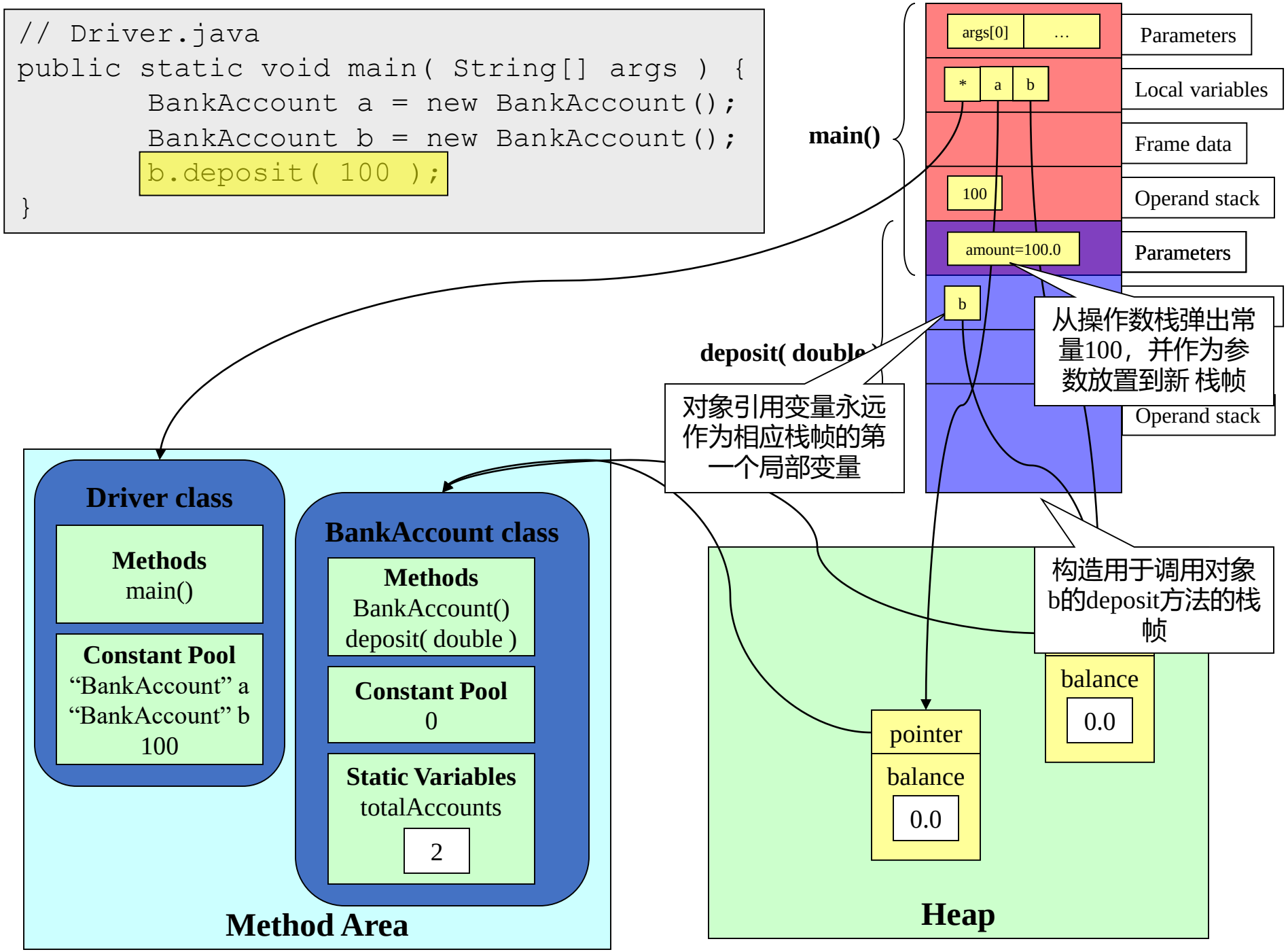

// Driver.java
public static void main(String[] args) {
BankAccount a = new BankAccount();
BankAccount b = new BankAccount();
b.deposit(100);
}
BankAccount类已经被加载，无需再次加载
main()
args[0]
…
Parameters
*
a
b
Local variables
Frame data
100
在heap常量区中创建指向BankAccount 的指针
BankAccount()
*
Local variables
Frame data
Operand stack
Driver class
Methods
main()
Constant Pool
“BankAccount” a
“BankAccount” b
100
BankAccount class
Methods
BankAccount()
deposit(double)
Constant Pool
0
Static Variables
totalAccounts
1
Method Area
针对BankAccount构造器调用创建的栈帧
pointer
pointer
balance
0.0
创建指向BankAccount类数据的指针
Heap


// BankAccount.java
private double balance;
private static int totalAccounts = 0;
public BankAccount() {
balance = 0;
totalAccounts++;
}
public void deposit(double amount) {
balance += amount;
}
因totalAccounts已经初始化了，无需再次初始化
main()
BankAccount()
args[0]
…
Parameters
*
a
b
Local variables
Frame data
100
Operand stack
Parameters
*
Local variables
Frame data
Operand stack
Driver class
Methods
main()
Constant Pool
“BankAccount” a
“BankAccount” b
100
BankAccount class
Methods
BankAccount()
deposit(double)
Constant Pool
0
Static Variables
totalAccounts
2
Method Area
totalAccounts增加1
该对象的balance变量被初始化默认值
该对象的balance被设置为 0
pointer
balance
0.0
balance
0.0
Heap

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



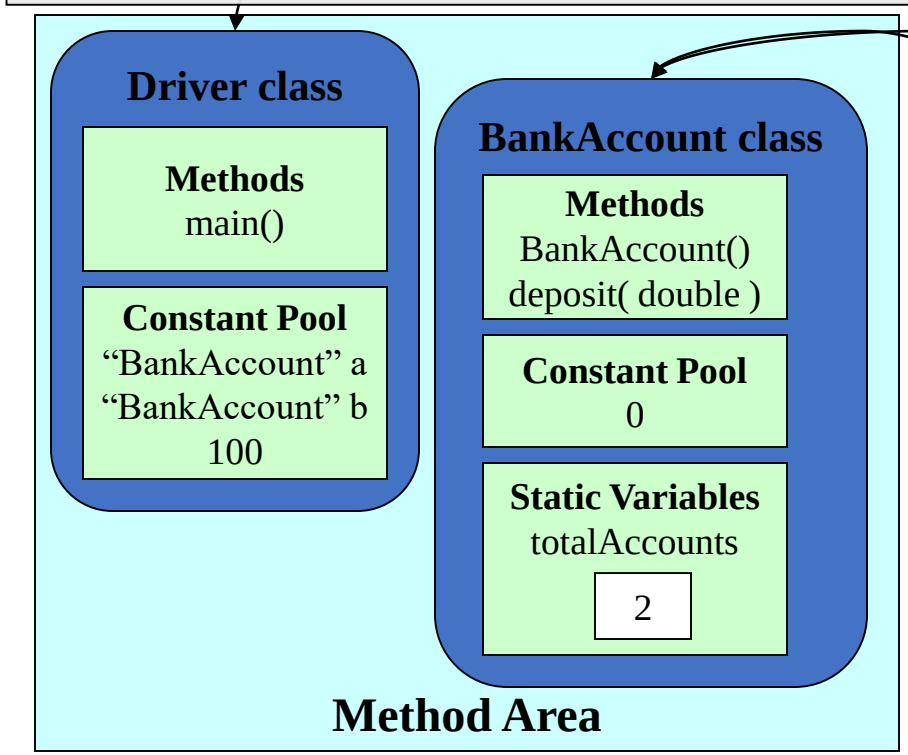
```
// Driver.java
public static void main( String[] args ) {
        BankAccount a = new BankAccount();
        BankAccount b = new BankAccount();
        b.deposit( 100 );
}
```

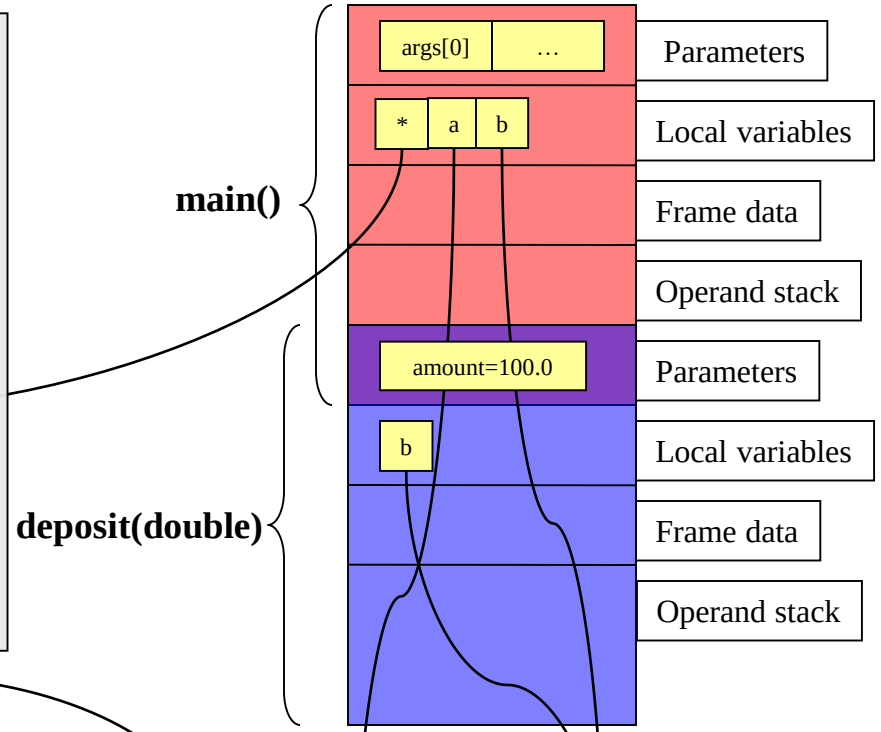


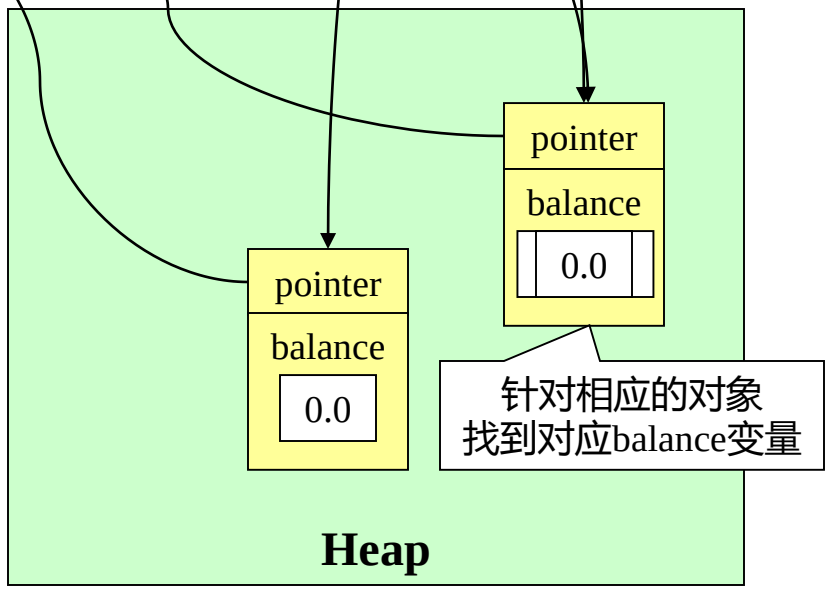
```
// BankAccount.java
private double balance;
private static int totalAccounts = 0;

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

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

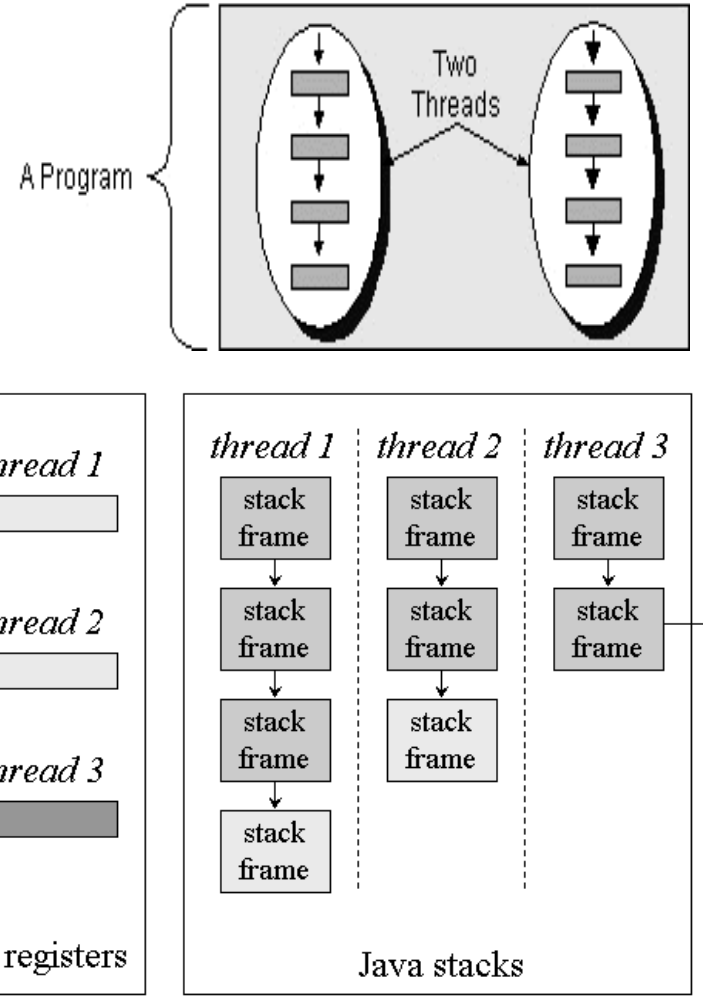






线程是什么

- 程序执行的本质是运行其指令，形成执行(控制)流(flow of control)
  - 执行流内的指令顺次运行
  - 任意时刻，一个执行流中只有一个指令在运行
  - 每个执行流都使用独立的栈来追踪其函数调用
- **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**封装了业务**数据**



Java多线程程序

- 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

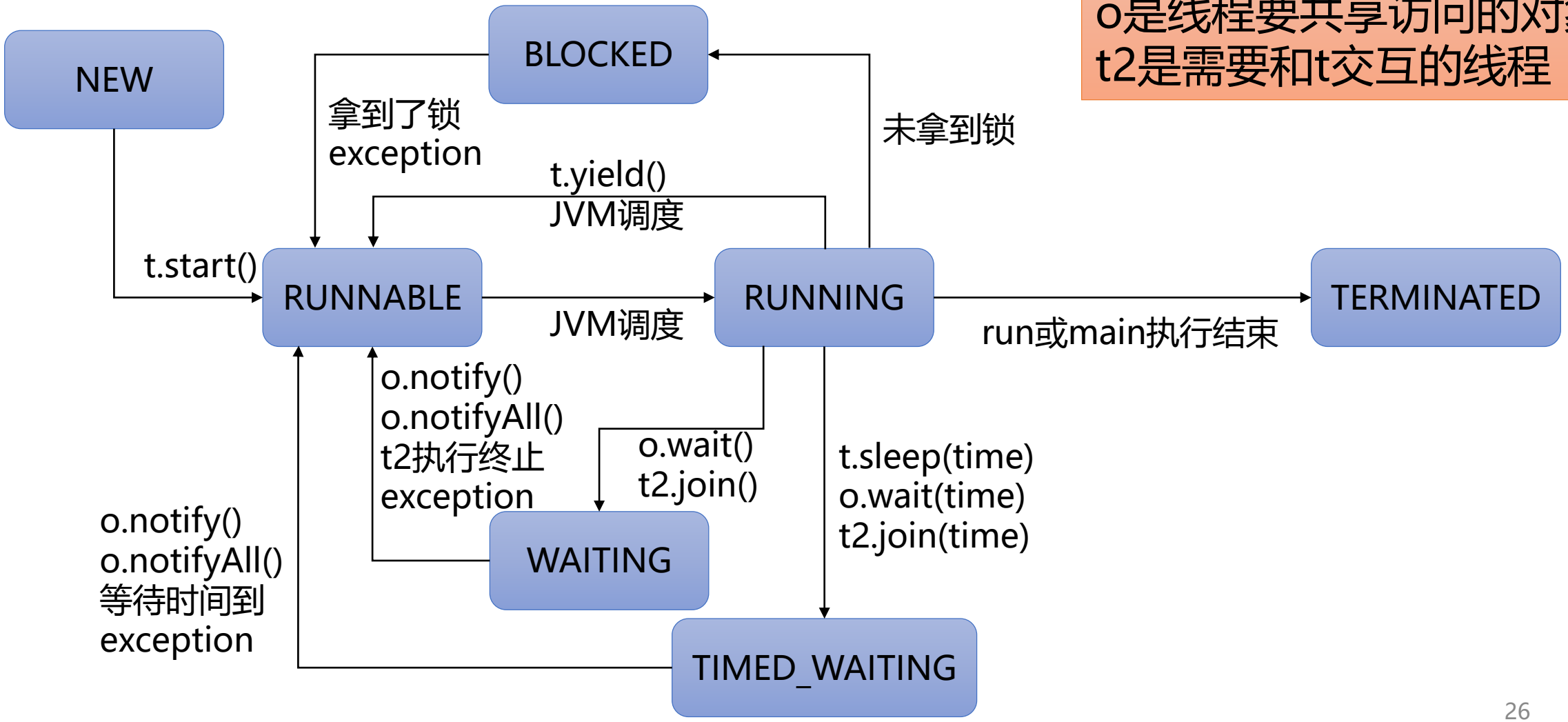
```
public void run() {
    try {   ...
        while (true) {      // 常规唤醒从这里继续执行
            if (有新任务) do the task;
            if (不再有新任务) break;
            sleep(...) or wait(...);    // 休息一会，让其他线程有机会执行
        }
        do something to finish;
    }
    catch (InterruptedException e) {  // 异常唤醒(即interrupt被调用)从这里继续执行
        ...  // 状态不太正常，善后处理...
    }
    do something to finish;
}
```

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

线程状态变化的控制机制

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



无数据交互关系的线程行为不确定性

```
public class SimpleThread extends Thread {
    public SimpleThread(String str) { super(str);  }
    public void run() {
        int i = 0;
        while (true) {
            System.out.println(i+" "+getName());
            i++;
            if (i==9) break;  //不再有新任务了
            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();
  }
}
```

交替具有不确定性，但是只看t1/t2，顺序是一定的

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

有数据交互关系的线程行为不确定性

Thread t1

```
a.deposit(amount=50) :  // deposit 1
    x = this.getBalance()  //1
    x += amount; //2
   this.setBalance(x) //3
```

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

Thread t2

```
b.deposit(amount=40) : // deposit 2
    x = this.getBalance()  //4
    x += amount; //5
    this.setBalance(x) //6
```

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？

会导致存入40.

- 经典场景：多个线程对共享对象进行读写
  - 读和写次序不确定（不知道当前数据是否为新数据）
  - 程序执行结果也出现了不确定
  - ➔**数据竞争**、读写**不一致**

如何处理线程行为的不确定性?

- 如果两个线程之间没有数据交互关系，不会影响最终执行结果
  - 否则可能会影响最终执行效果
  - 不确定性：不知道是否/何时有新数据
- 有可能长时间没有新数据到达
  - 长时间轮询导致**线程空转**，浪费CPU资源
- 当有新数据可用时
  - 告诉JVM来**唤醒**其他等待（该数据）的线程：notify/notifyAll
- “共享数据是否可用”这个不确定性必须控制
  - 让谁来控制?

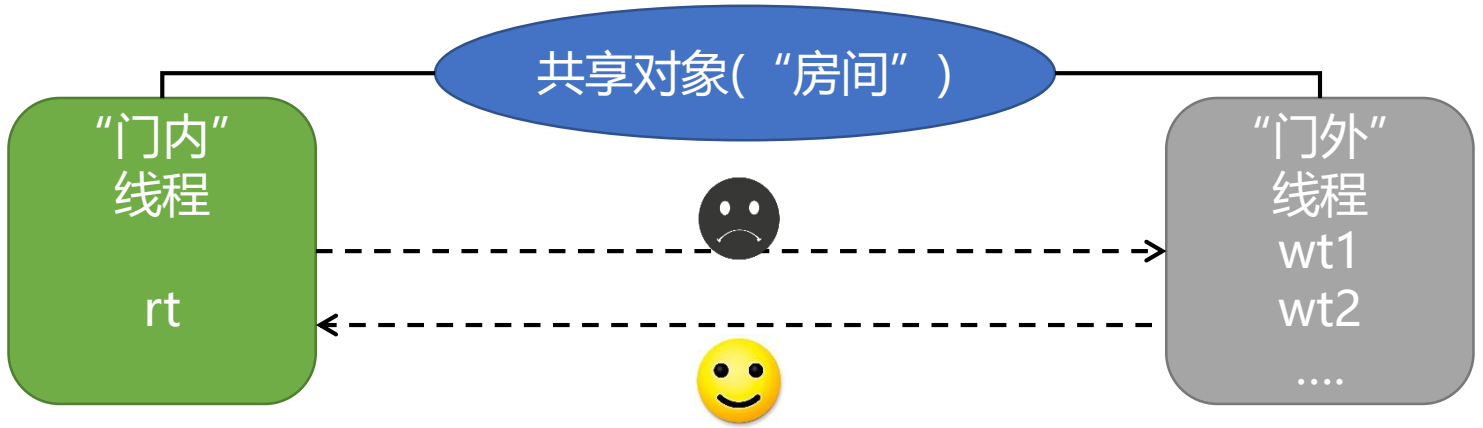
➔ 循环查询: while (true){...}

➔避免空转的查询: while (true){ this.sleep()/o.wait()...}

➔让这个**共享对象**自己来控制!

共享对象应如何控制?

- 导致**是否有新数据**不确定的原因是**多个线程同时访问共享对象**
- ➔**互斥**是根本：任何时刻只允许一个线程访问
  - **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**来让JVM调度等待线程



同步控制块是如何发挥作用的

- 每个synchronized block都关联到一个监控对象(monitor)
  - JVM确保每个对象只有一个lock
  - JVM确保拿到montior.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() {
        …
   }
}
```

判断：手中的锁能否进到m2中。而锁是针对this的，所以可以调度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原则
  - 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t.sleep，o.wait来进行**避免空转的轮询**
- 共享对象代码
  - 建立synchronized临界区
  - 设置**是否有新数据**的状态标识
  - 退出临界区前notify/notifyAll其他线程

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**

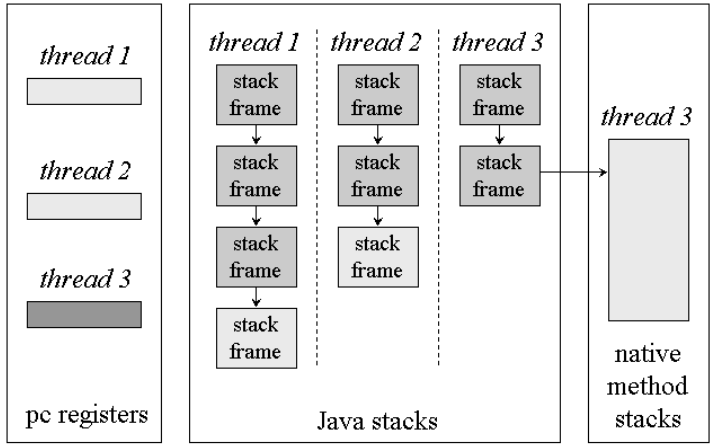
```
class CustomerDb3 {
    private final List<Customer> customers = new ArrayList();
    public double countOrderValue (Customer c){
        ArrayList<Order> orders;
        synchronized (c){
            orders = c.getOrders();
        }
        double total = 0.0;
        for(Order o : orders)
   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

c是**共享对象**，c所管理的orders也是**共享对象**！

线程类的基本设计框架

- 实现一个独立和完整的算法/功能
  - run方法（相当于main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
- 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

线程栈栈底存的是什么？



线程类的基本设计框架

- 通过共享对象与其他线程交互
  - 通过锁来确保任何时候只有一个线程访问共享对象
    - 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(notify/notifyAll)**其他等待线程
    - 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();
}
```

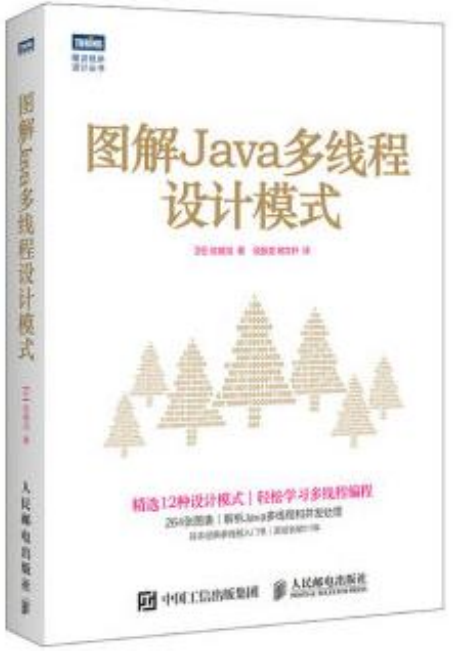
```
...main(String[] args){
    ...
    t1.start();
    t1.run();
}
```

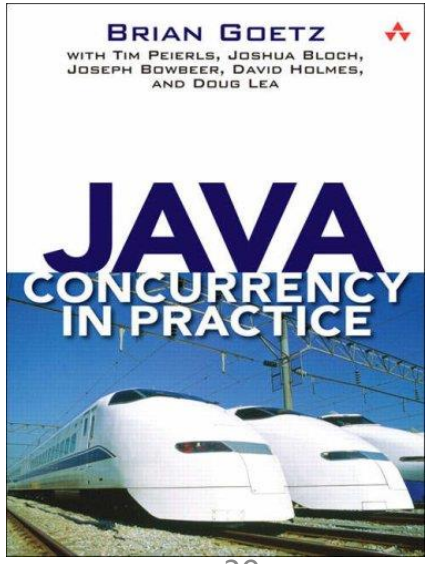
0或者1

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

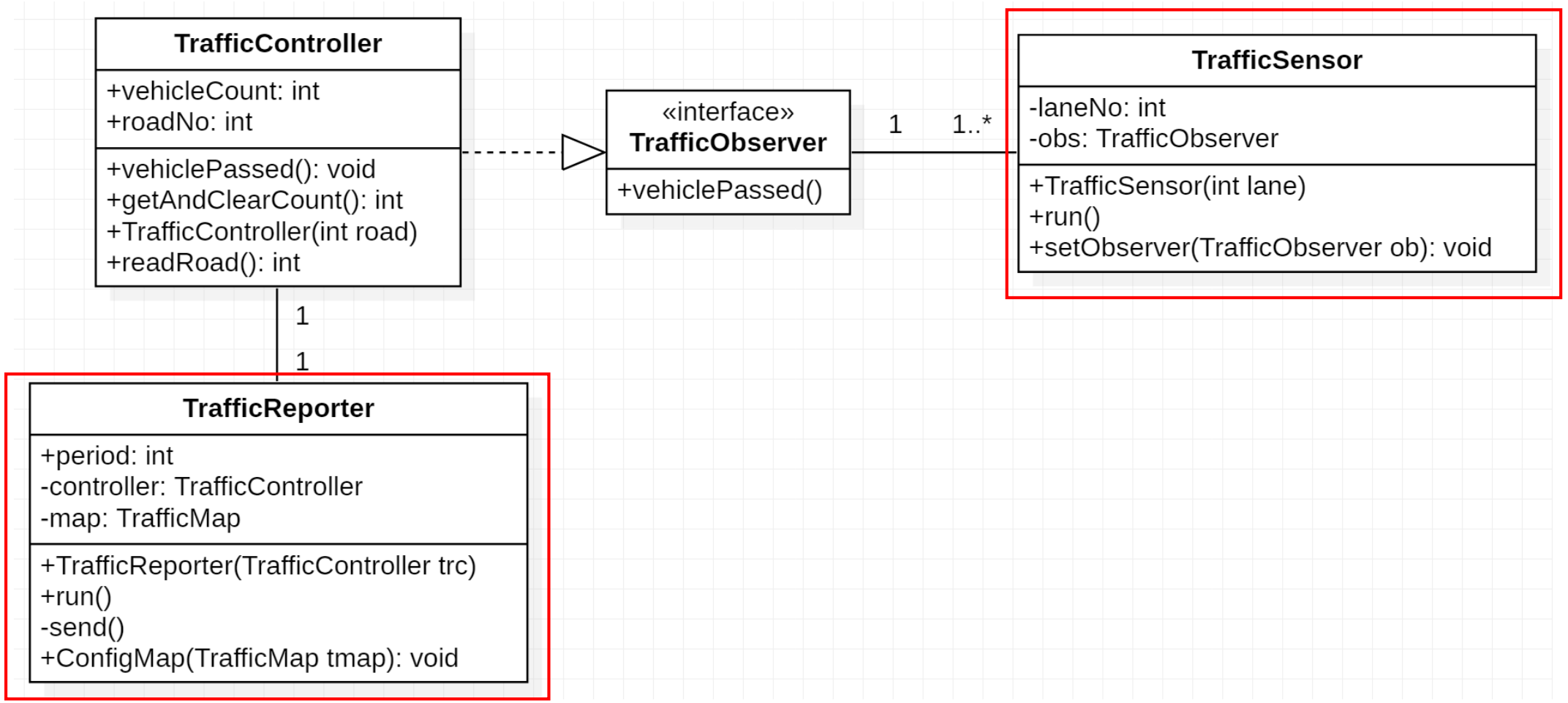
多线程交互模式：线程互斥访问

- 多个线程互斥访问共享对象
  - 要点：把共享对象的方法进行synchronized保护
- 案例分析
  - 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？

错误，因为getAndClearCount中访问了下一个层次的vehicleCount

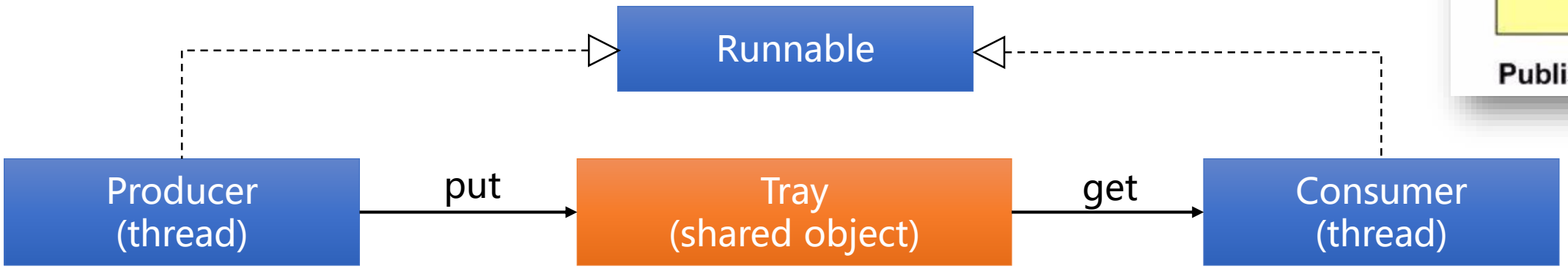
```
public class TrafficController {
    private int vehicleCount;
    private int roadNo;
    public TrafficController(int road){
        roadNo = road;
        vehicleCount = 0;
    }
    public synchronized void vehiclePassed() {
    //all sensors will call
        vehicleCount ++;
    }
    public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

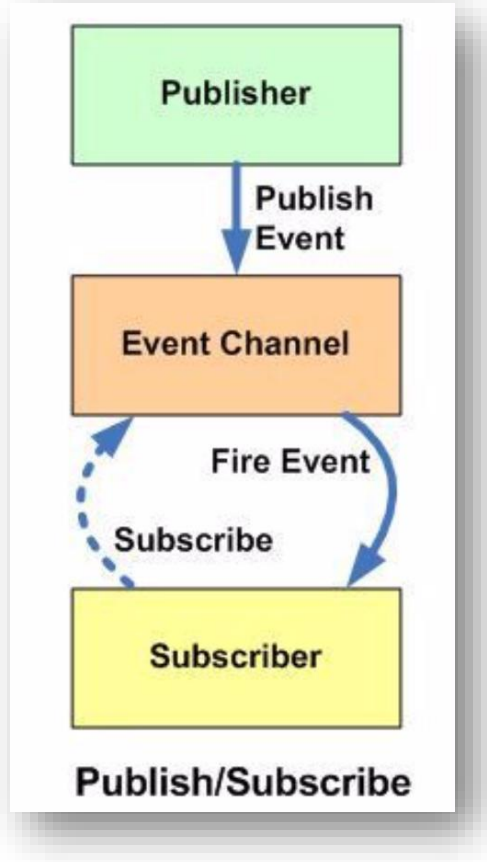
多线程交互模式：线程互斥访问

- 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线程，一个controller对象和一个reporter线程
  - 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

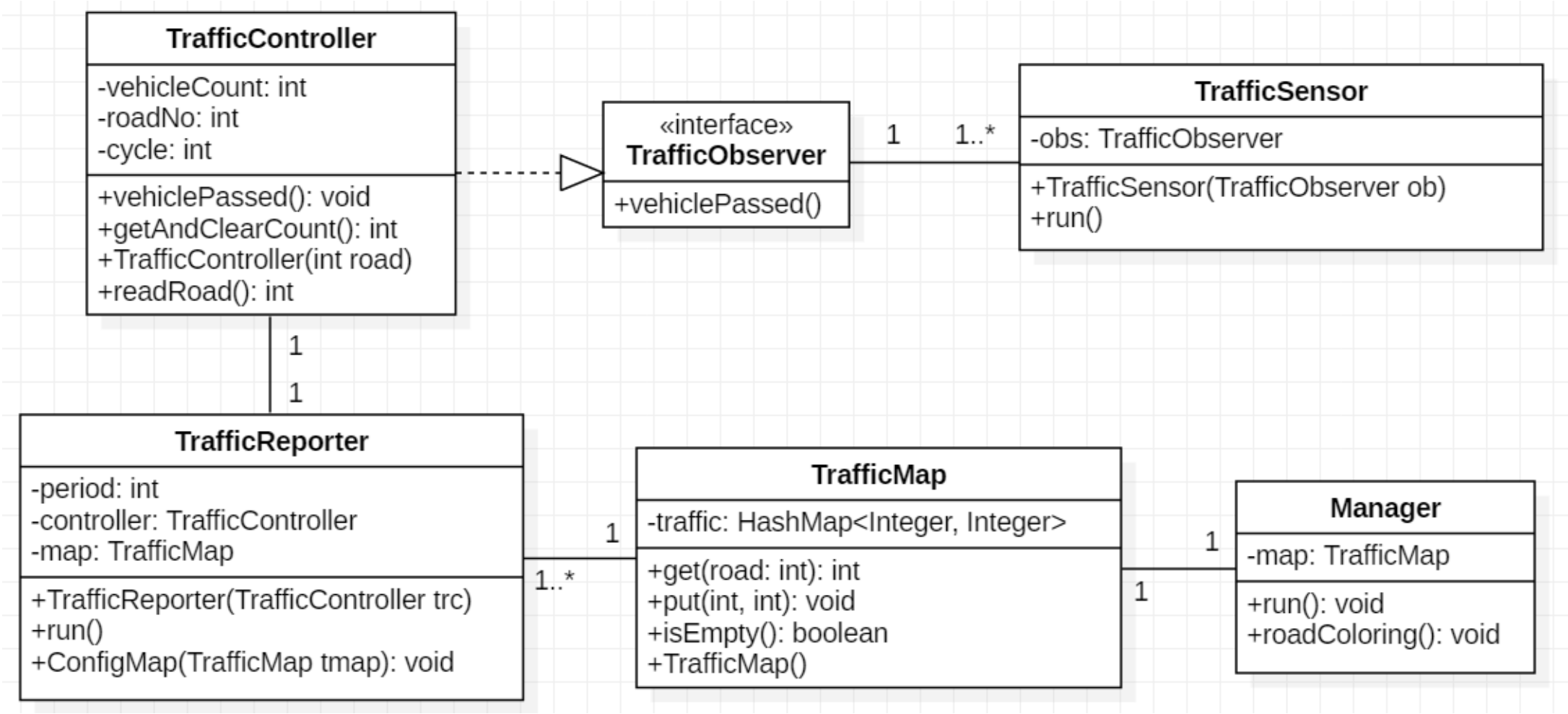
线程交互模式：生产者消费者

- 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; //共享对象
    private TrafficMap map; //共享对象
    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; //共享对象
    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);
            }
        }
    }
}
```

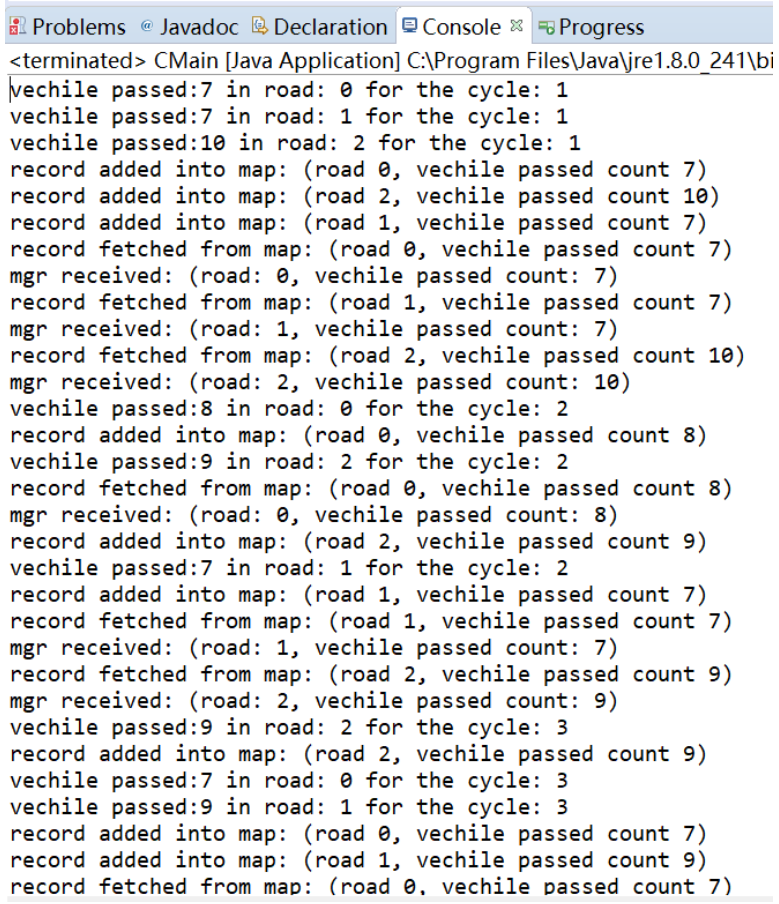
线程交互模式-可伸缩的传送带

- 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：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-**<*controller*>**-Reporter-**<*map*>**-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



单线程程序与多线程程序的对比

对比项	单线程程序	多线程程序
执行流数目	只有一个	一到多个
流程控制	方法调用、分支控制	方法调用、分支控制； 线程的创建、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

本周作业分析

- 多线程电梯系统
  - 电梯各项参数静态不可变（运行速度、开关门耗时、限乘人数、楼层数、初始位置）
  - 6部电梯
  - 提供输入和输出处理包，**聚焦于线程交互控制和基本的电梯调度**
- 输入输出定义
  - 输入：[时间戳]乘客ID-FROM-x-TO-y-BY-z：乘客通过电梯z从x层到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，确保所有有效的乘客请求都得到了正确响应
  - 你的调度应至少和ALS策略的调度性能相当
  - 性能分计算依据：完成请求的耗时和电梯“耗电量”

设计建议

- 有哪些线程类?
  - 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
  - 2、请求输入装置、电梯和调度器是线程，请求队列是共享对象
    - 这三个线程之间的生产-消费关系是什么?
- 如何调度电梯?
  - 分析**候乘表**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
  - ➔候乘表类、调度策略类