第五讲
Java对象运行机制与多线程

OO课程组2019

北京航空航天大学计算机学院

内容摘要

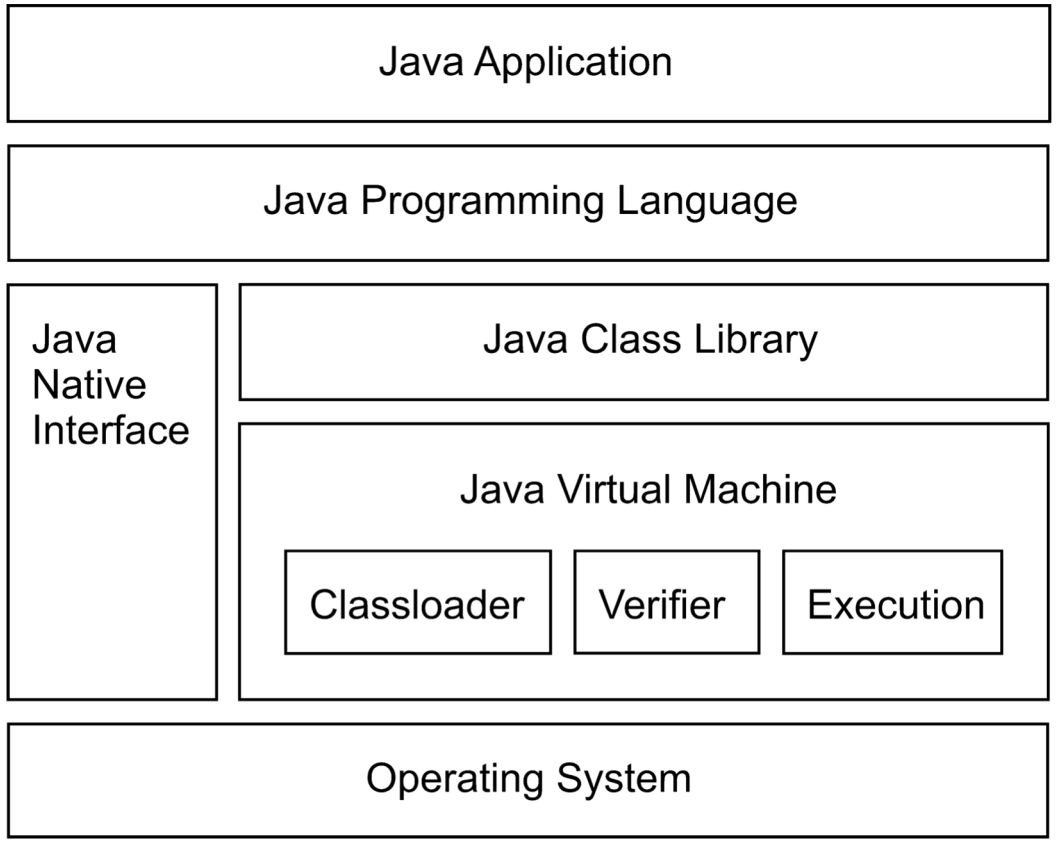
- Java系统概览
- Java应用如何执行
  - JVM基本结构
  - JVM内存划分
  - 对象方法调用
- Java应用的多线程处理初步介绍
- 多线程设计模式选讲
- 作业解析

Java系统概览

你编写的Java程序

Java编译器

本地程序调用接口

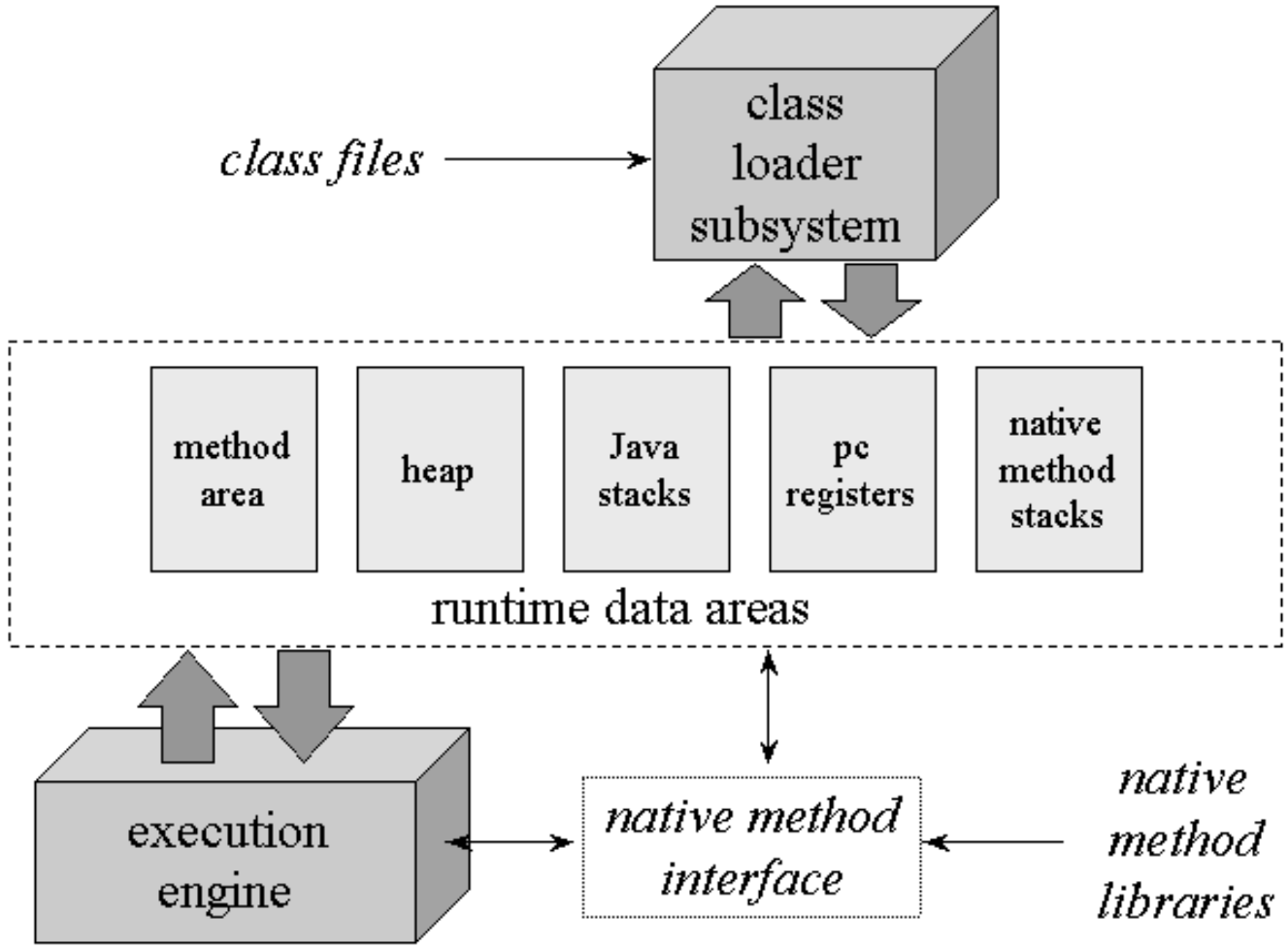


Java类库程序包

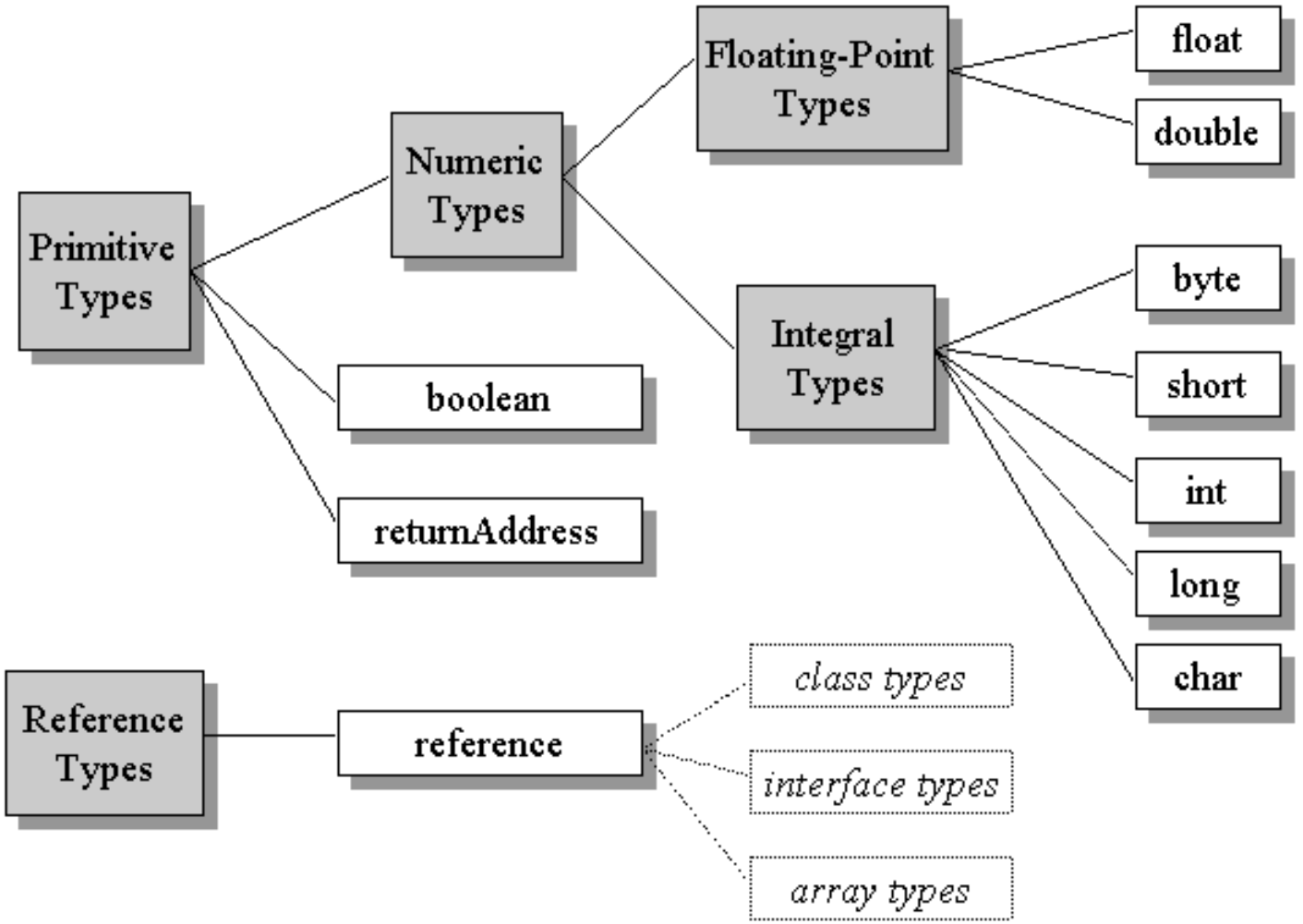
Java程序执行虚拟机

操作系统

Java虚拟机(Java Virtual Machine)

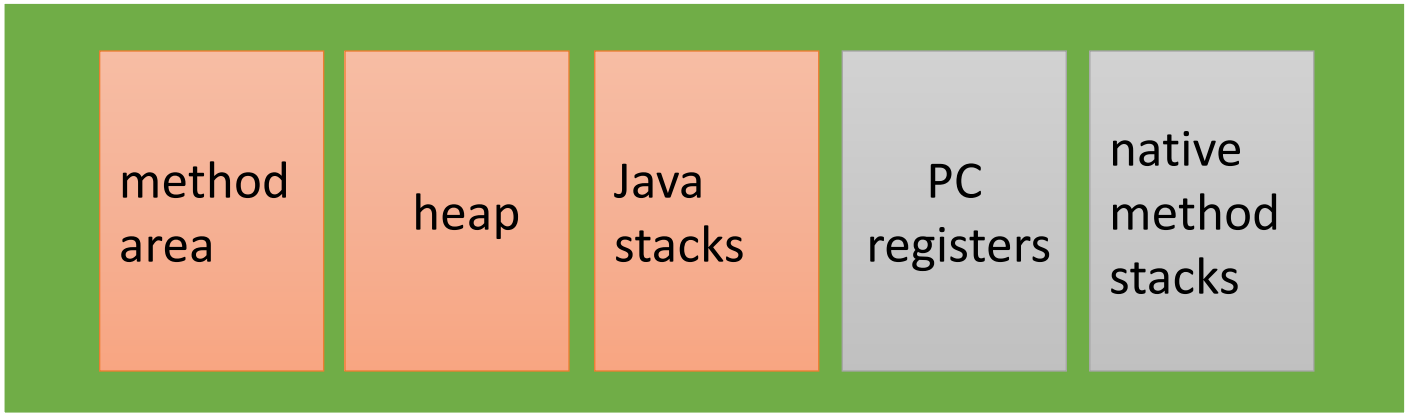


JVM中用到的数据类型



JVM内存区域划分

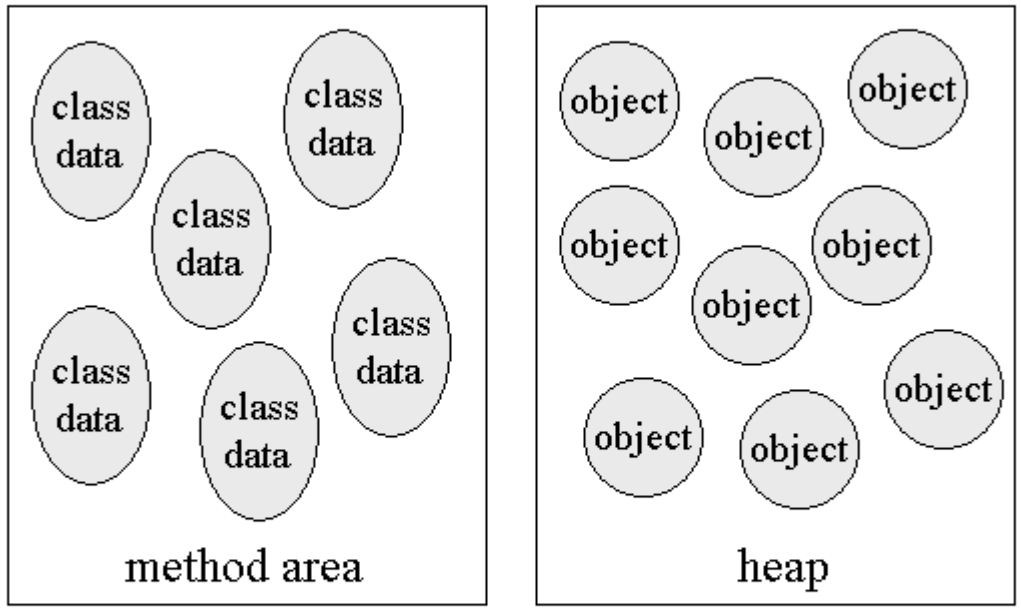
- Java程序运行时由一到多个线程**组成**
- 每个线程都有自己的栈
- 所有线程共享同一个堆



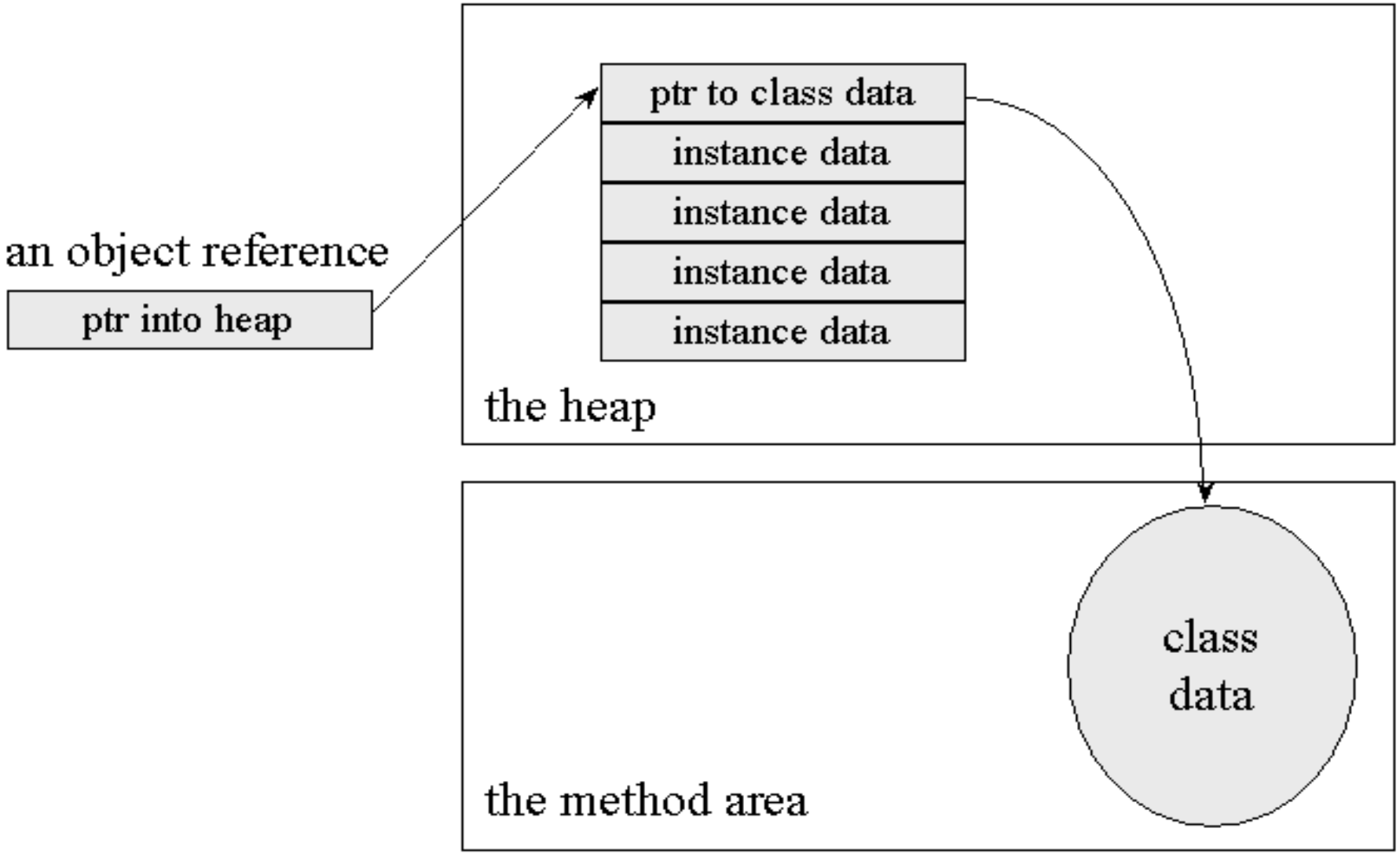
- Method area
  - Class description
  - Code
  - Constant pool
- Heap
  - Objects and Arrays
  - Shared by all threads
  - Garbage collection
- Stack
  - Invocation frame
  - Local variable area
  - Operand stack

JVM内存区域划分

- Method内存区域
  - 保存class信息
  - 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



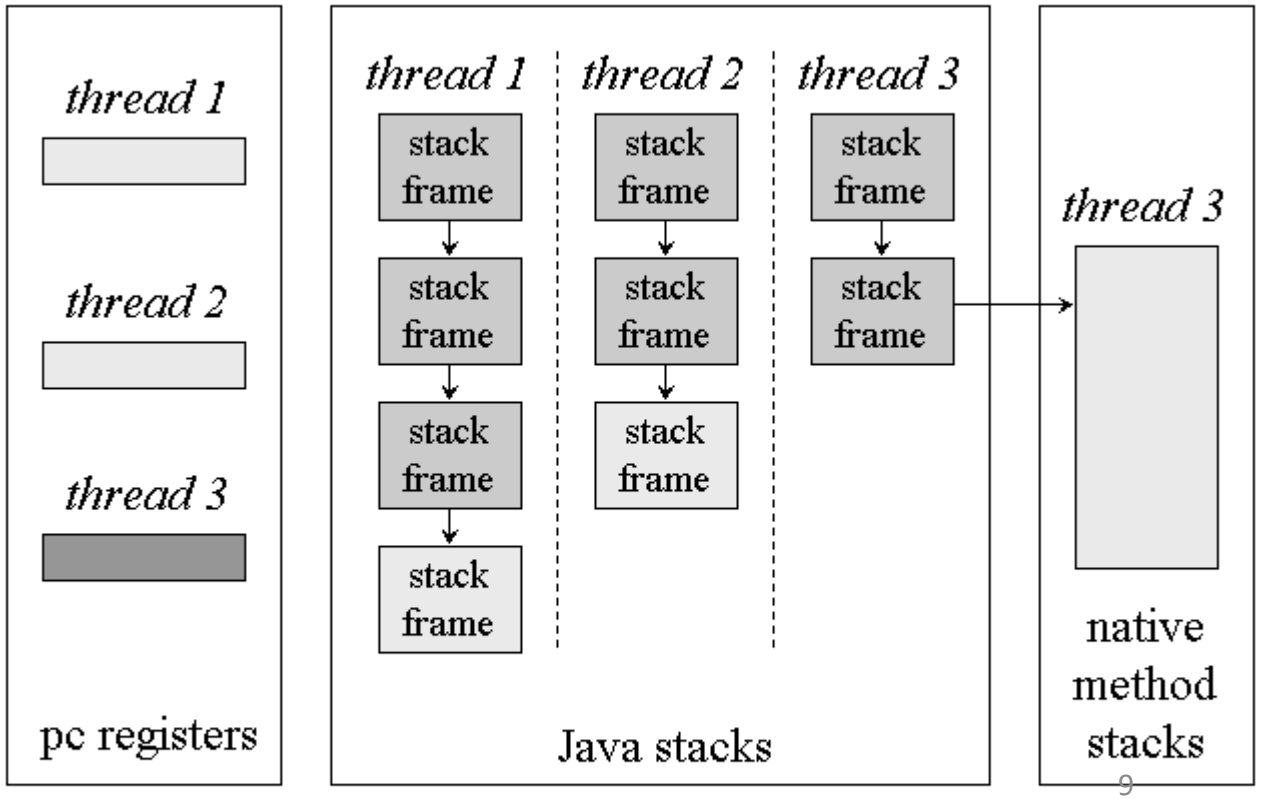
堆中的对象表示


an object reference
ptr into heap
ptr to class data
instance data
instance data
instance data
instance data
the heap
class
data
the method area

JVM中的内存划分

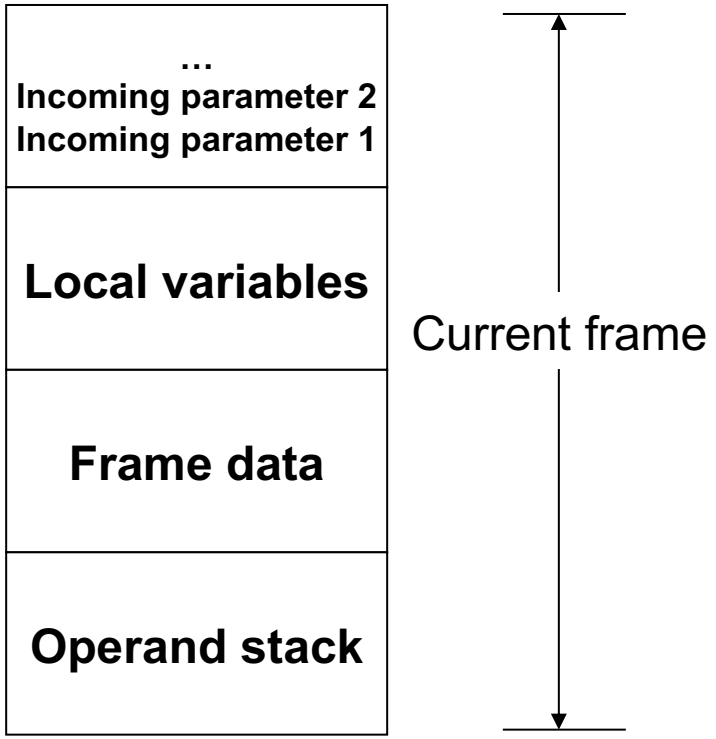
- 栈内存区
  - 每个线程都拥有专属的栈内存，用以追踪执行路径
  - 栈由栈帧组成，顶栈帧描述线程的当前执行状态
  - JVM对栈帧进行push和pop操作

线程栈栈底存的是什么？



栈帧结构

- 方法输入参数
- 方法局部变量
  - 组织成一个数组
  - 通过下标来访问局部变量元素
- 栈帧数据(Frame Data)
  - 到类常量区(method area)的引用
  - 方法调用返回
    - 没有触发异常
    - 把返回值放置到上一帧中
  - 异常处理分派(dispatch)
- 操作数栈(Operand Stack)
  - 组织成数组
  - 通过入栈和出栈来访问
  - 始终处于栈顶
  - 为方法中的各种计算操作提供了工作空间

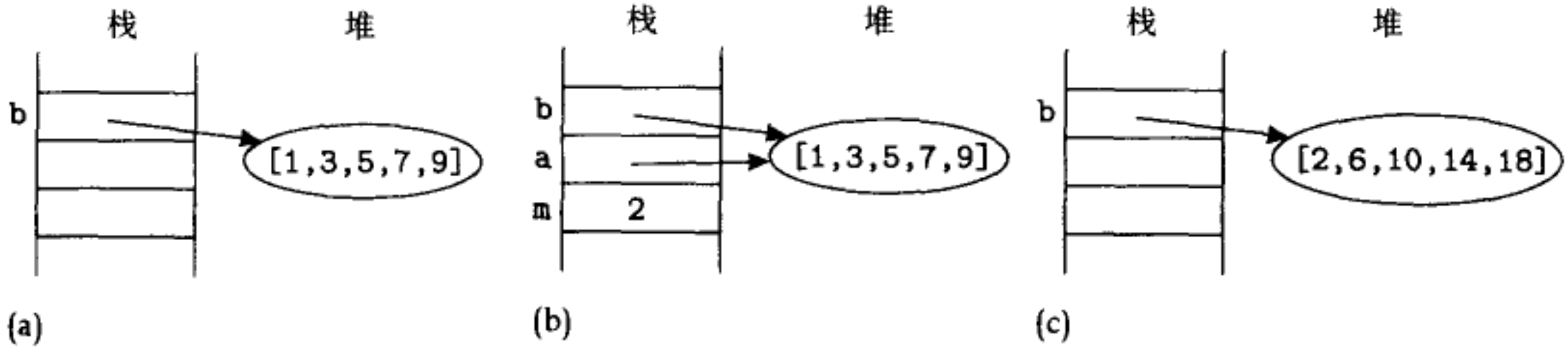


对象方法调用时发生了什么

- 比如调用e.m(...)
  - 首先，通过e获得相应的对象
  - 然后获得实参的取值
  - 创建一个栈帧，并push到栈顶
  - 根据对象的class信息(分派机制)去调用具体的方法m

```
public static void multiples (int [ ] a, int m) {
    if (a == null) return;
    for (int i = 0; i < a.length; i++) a[i] = a[i]*m;
}
```

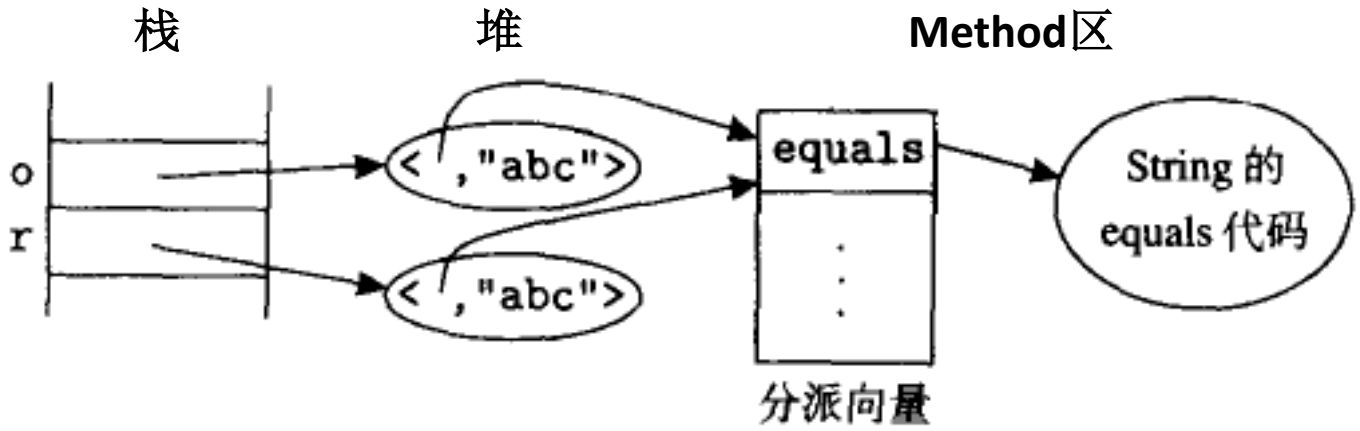
```
int [ ] b = {1,3,5,7,9};
Arrays.multiples(b, 2);
```



对象方法调用时发生了什么

- 每个对象的数据中包括了一个指向分派向量(dispatch vector)的引用
  - 分派向量提供了该对象所有方法的入口，存放在Method内存区的class数据中
  - 一个对象可以通过其创建时的类型或者其父类型来访问，但是类型转换不会改变对象中保存的分派向量

```
String t = "ab";
Object o = t + "c"; // concatenation
String r = "abc";
boolean b = o.equals(r);
```



Java程序运行时的内存状态变化

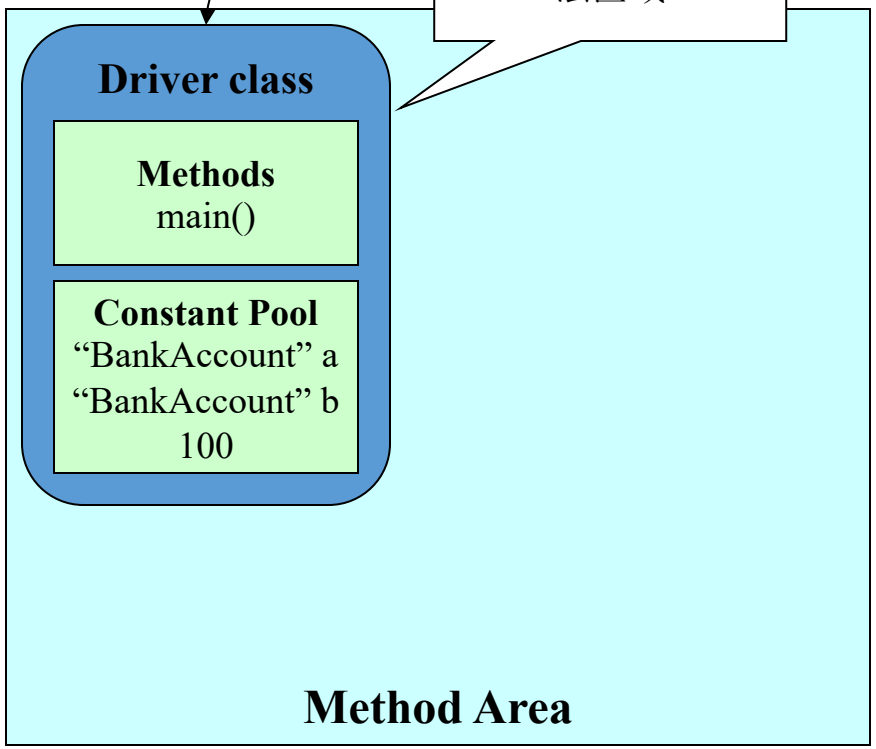
```
public class BankAccount {
    private double balance;
    private static int totalAccounts = 0;
    public BankAccount() {
        balance = 0;
        totalAccounts++;
    }
    public void deposit( double amount ) {
        balance += amount;
    }
}
```

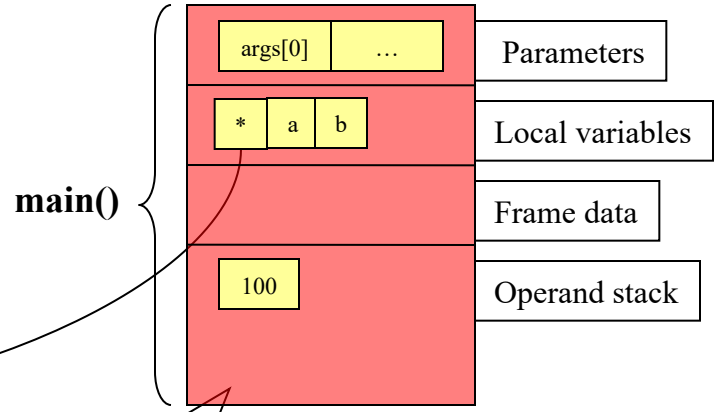
```
public class Driver {
    public static void main( String[] args ) {
        BankAccount a = new BankAccount();
        BankAccount b = new BankAccount();
        b.deposit( 100 );
    }
}
```

```
// In command prompt
java Driver

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

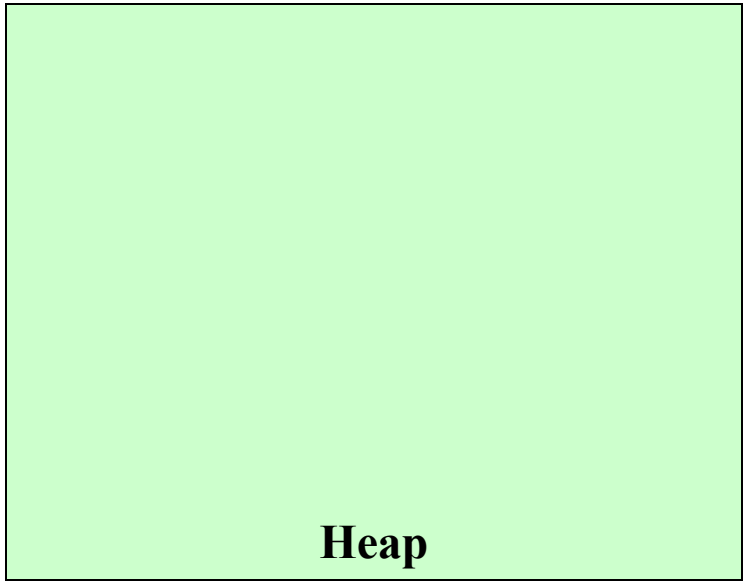
```
public class Driver {
    public static void main( String[] args ) {
        BankAccount a = new BankAccount();
        BankAccount b = new BankAccount();
        b.deposit( 100 );
    }
}
```

ClassLoader把Driver类加载到方法区域

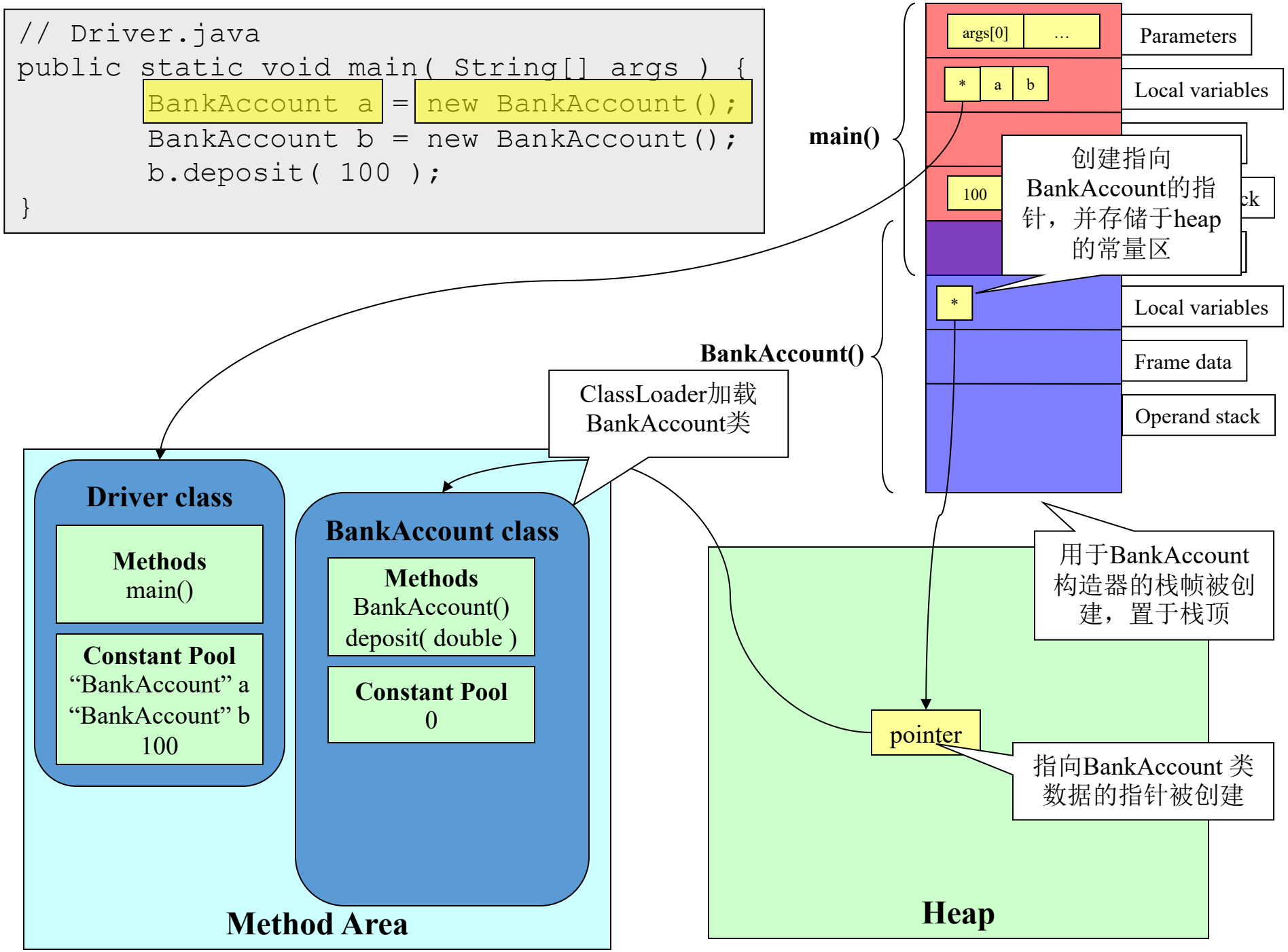




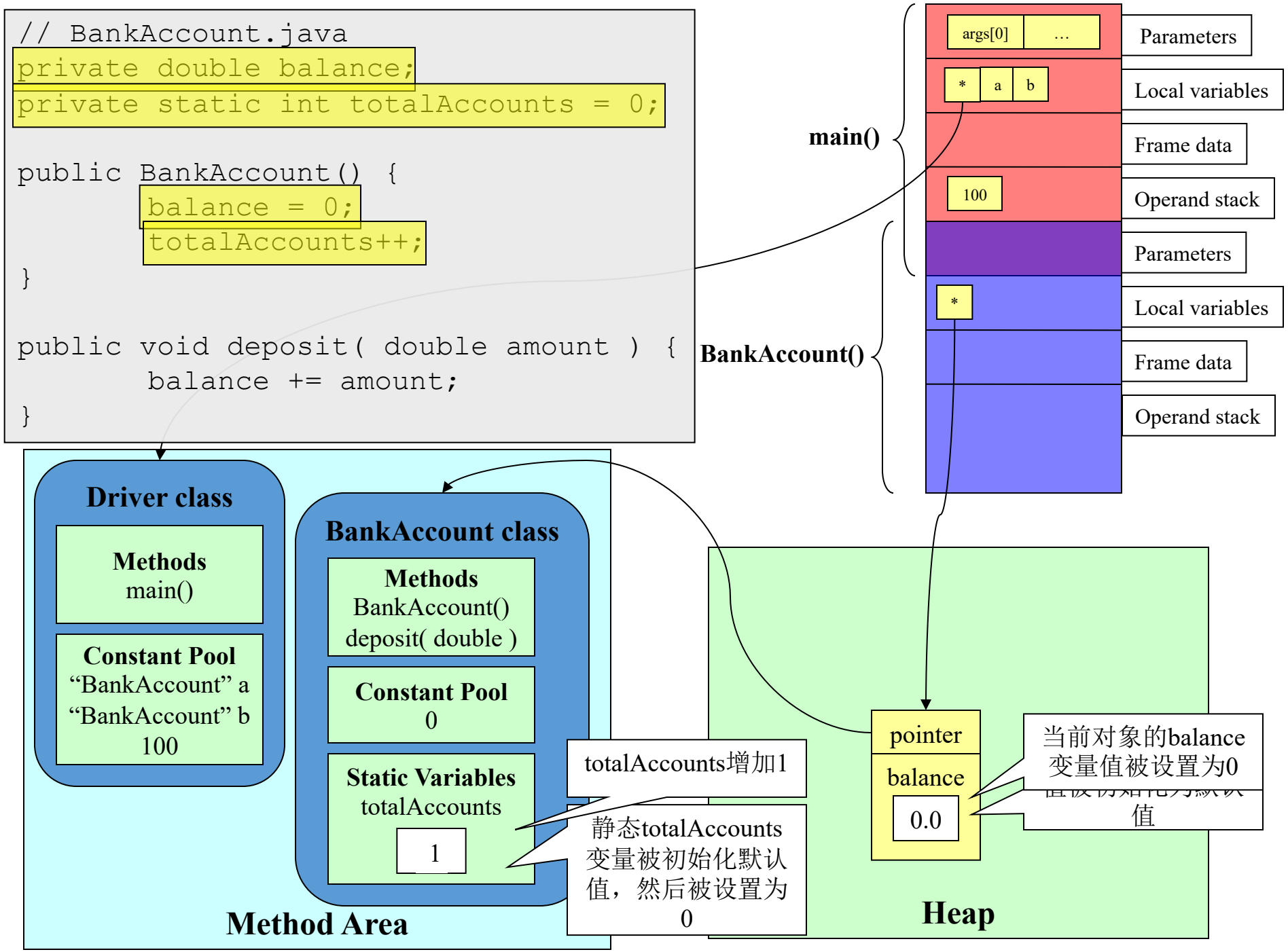
针对main()方法的栈帧被创建，并处于栈顶。*为对当前对象对应class的引用。



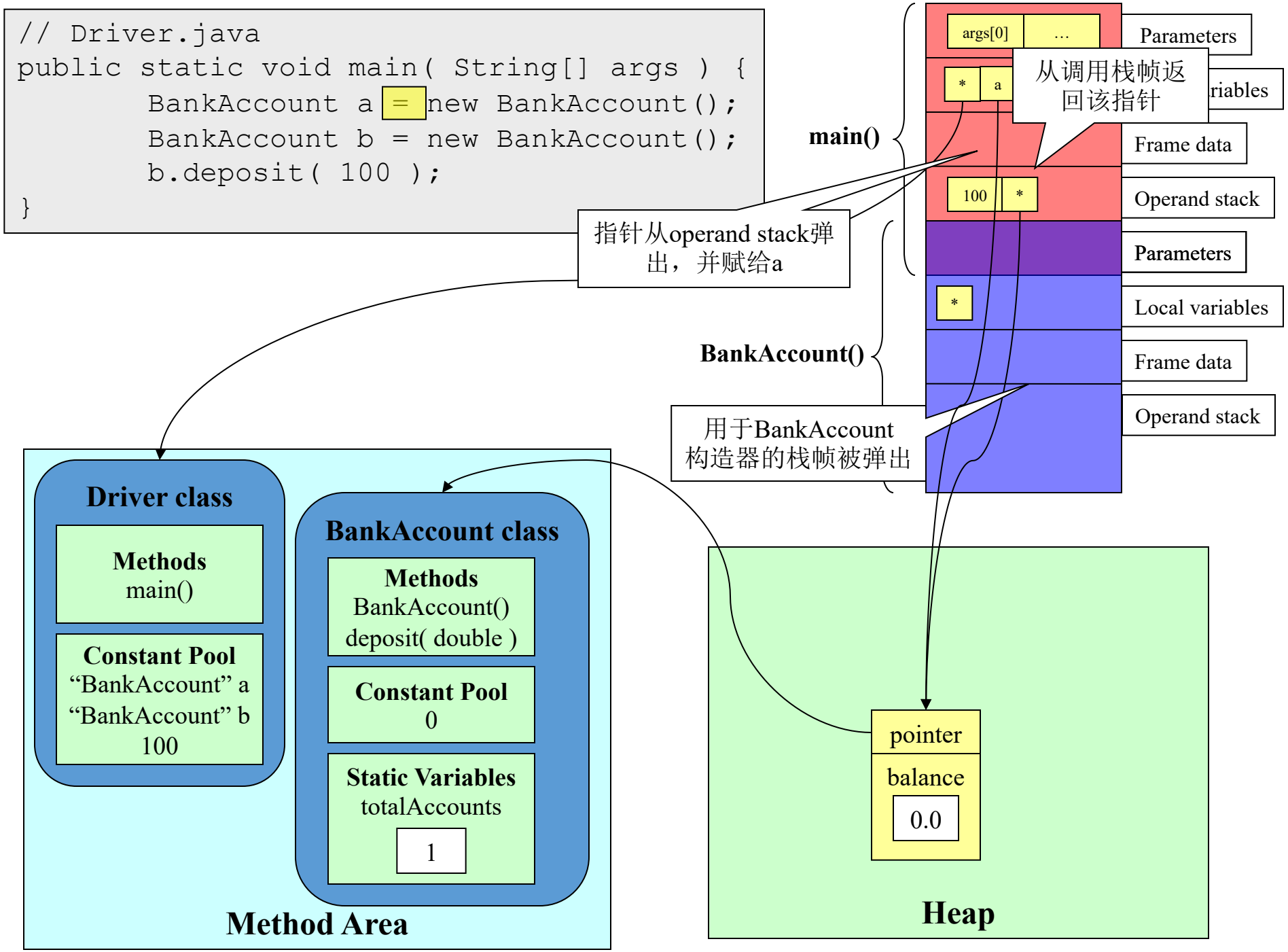
```
// Driver.java
public static void main( String[] args ) {
        BankAccount a = new BankAccount();
        BankAccount b = new BankAccount();
        b.deposit( 100 );
}
```

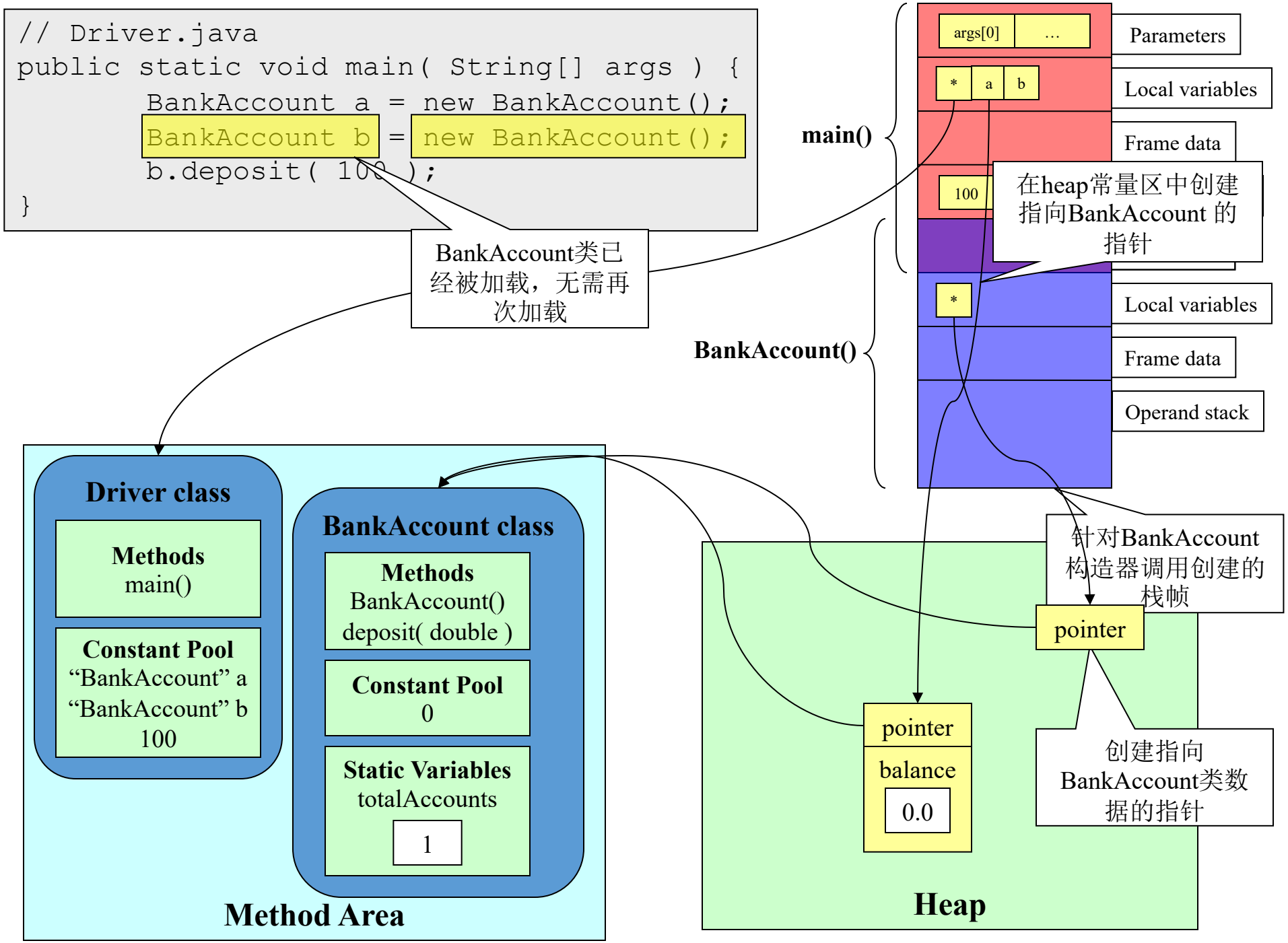


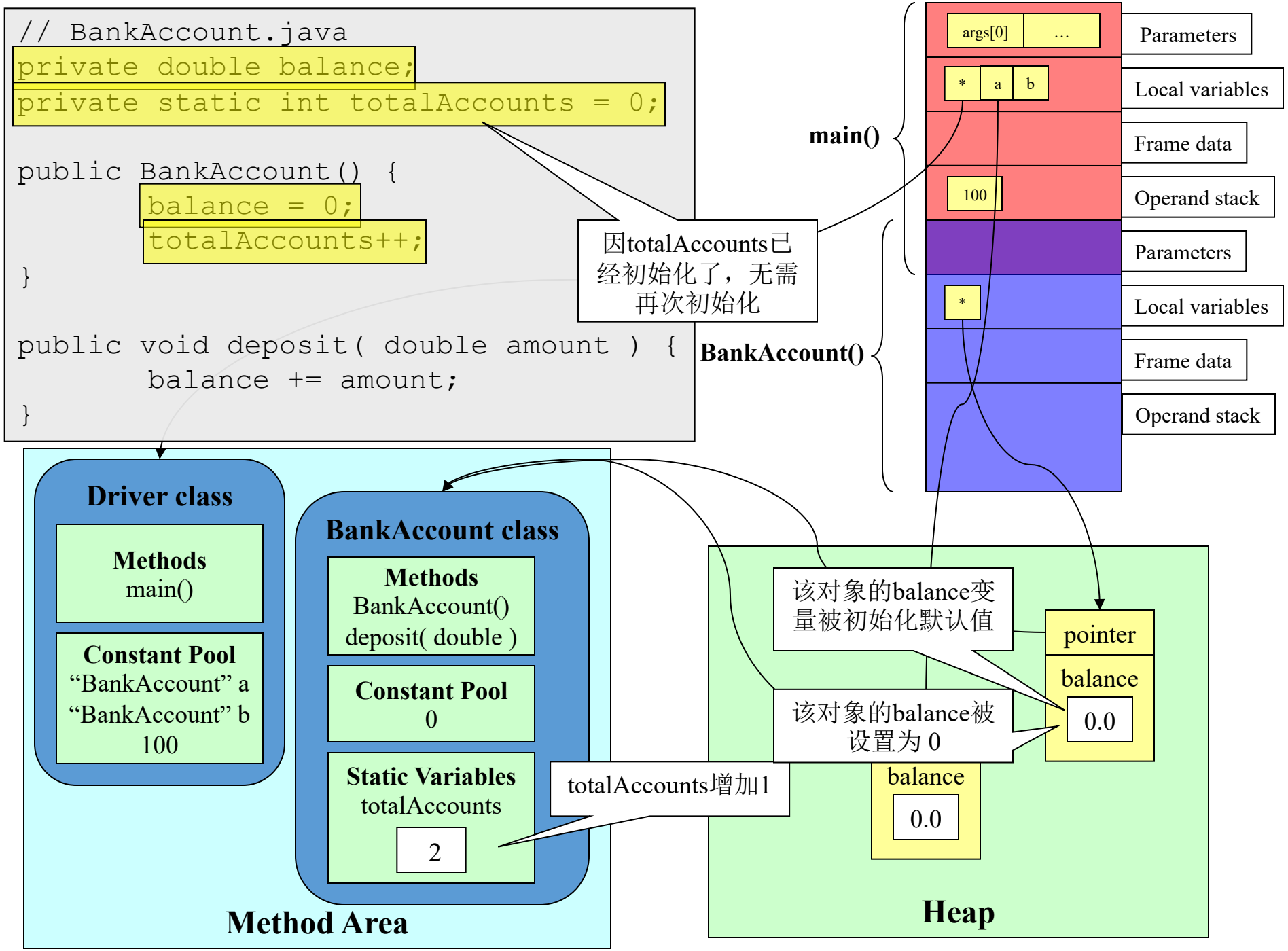
```
// BankAccount.java
private double balance;
private static int totalAccounts = 0;

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

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



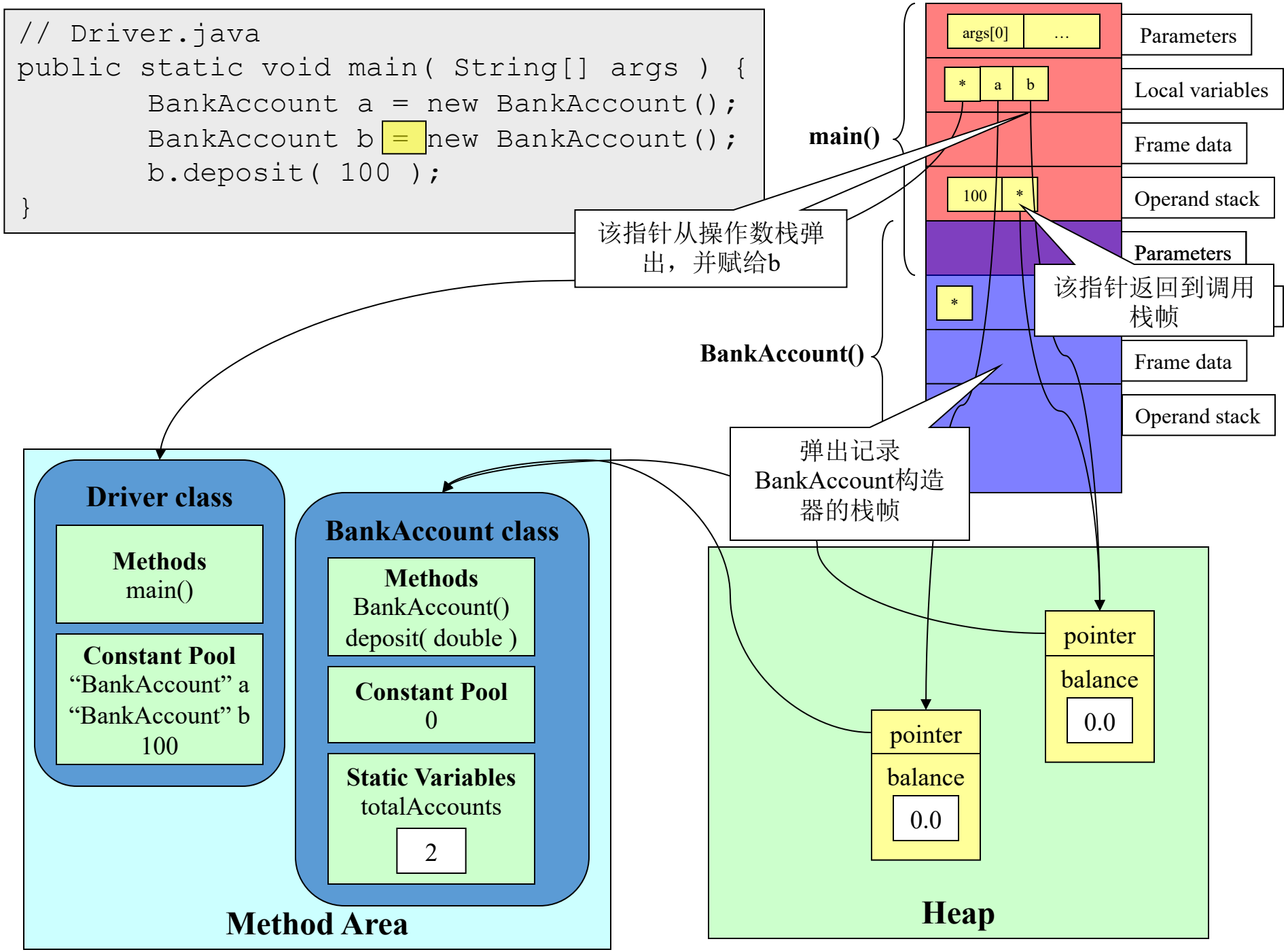
```
// Driver.java
public static void main( String[] args ) {
        BankAccount a = new BankAccount();
        BankAccount b = new BankAccount();
        b.deposit( 100 );
}
```



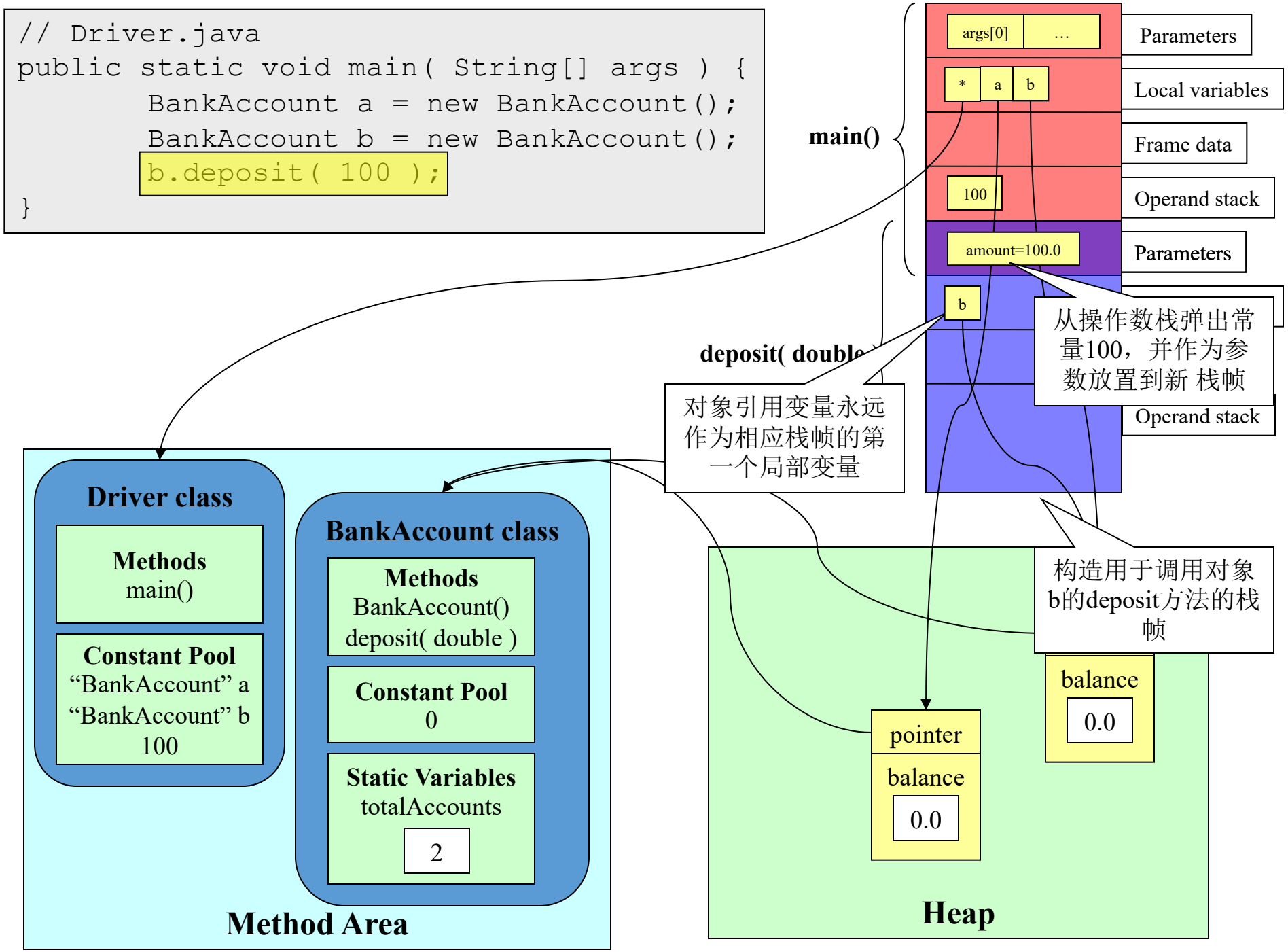

// Driver.java
public static void main(String[] args) {
BankAccount a = new BankAccount();
BankAccount b = new BankAccount();
b.deposit(100);
}
BankAccount类已经被加载，无需再次加载
main()
args[0]
…
Parameters
*
a
b
Local variables
Frame data
100
在heap常量区中创建指向BankAccount 的指针
BankAccount()
*
Local variables
Frame 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

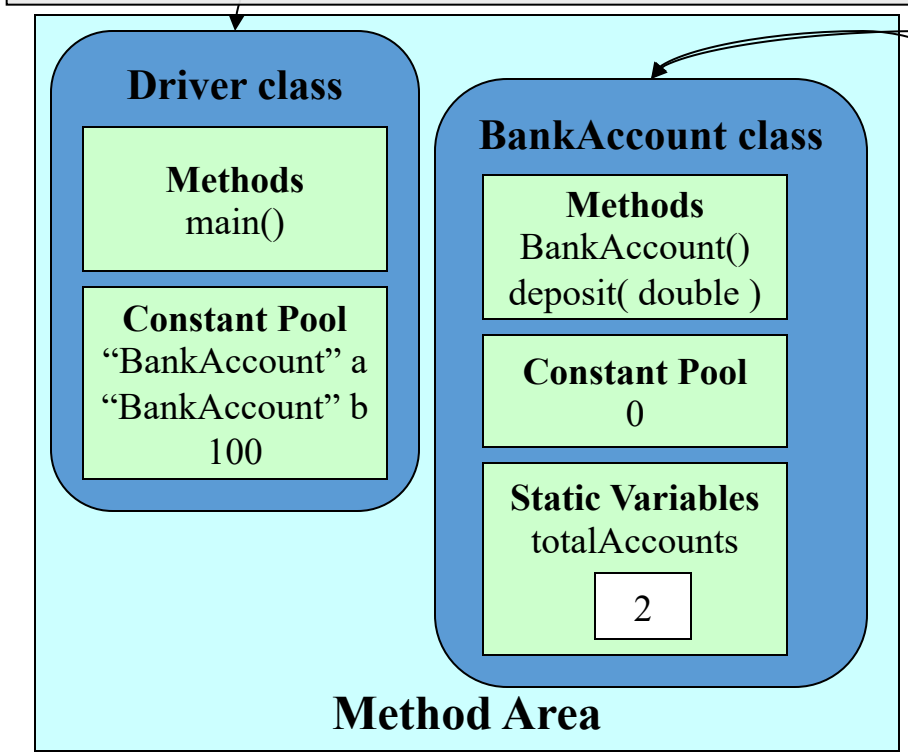
```
// Driver.java
public static void main( String[] args ) {
        BankAccount a = new BankAccount();
        BankAccount b = new BankAccount();
        b.deposit( 100 );
}
```

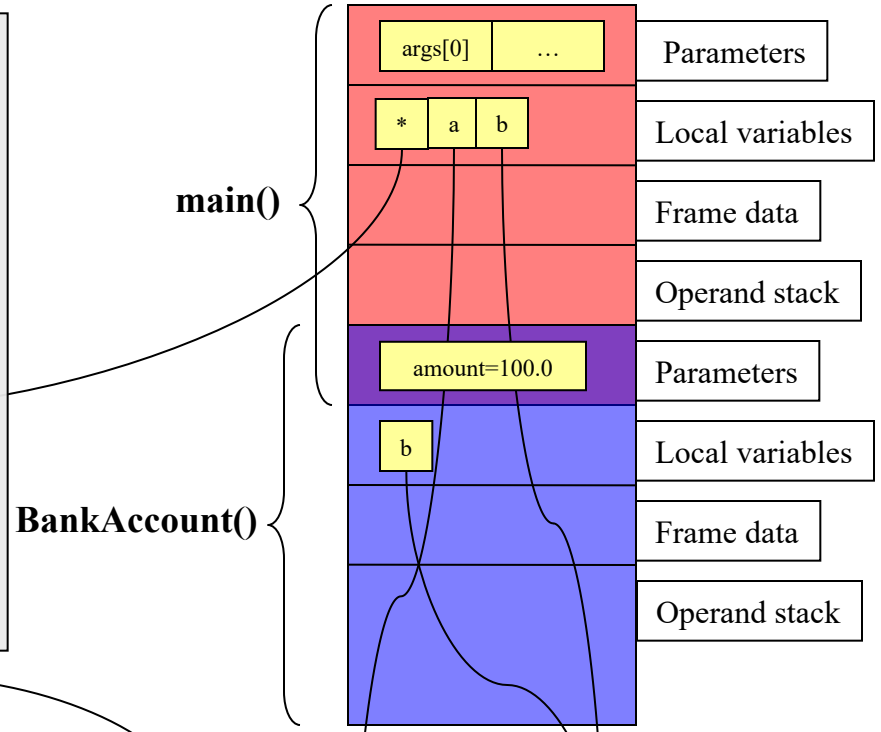


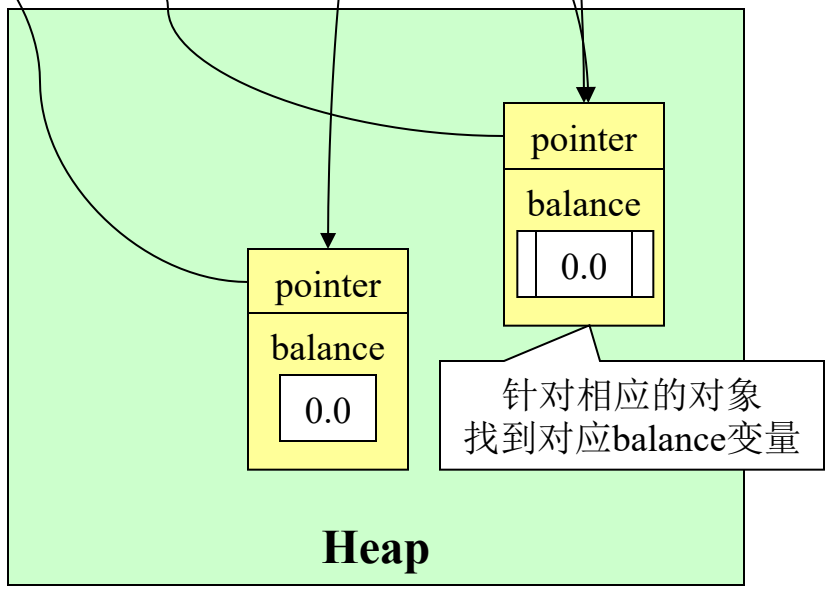
```
// Driver.java
public static void main( String[] args ) {
        BankAccount a = new BankAccount();
        BankAccount b = new BankAccount();
        b.deposit( 100 );
}
```



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

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

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







多线程处理

- JVM采用多线程来管理Java应用程序的运行状态
  - 一个Java程序----一个JVM实例----一个主线程
  - Java程序创建的线程----JVM栈帧进行管理
- 对象独立同时运行
  - 自成一体
  - 运行时需要彼此交互：调用、消息传递、共享数据
- 有时对象之间可采用松弛的“异步”交互方式
  - 通知对方自己“做了什么”或者“状态发生了改变”(下载对象与界面显示对象)→消息传递
  - 一边进行业务处理，一边通过共享对象来传递重要信息(调度对象与电梯对象)→共享数据

为什么需要多线程？

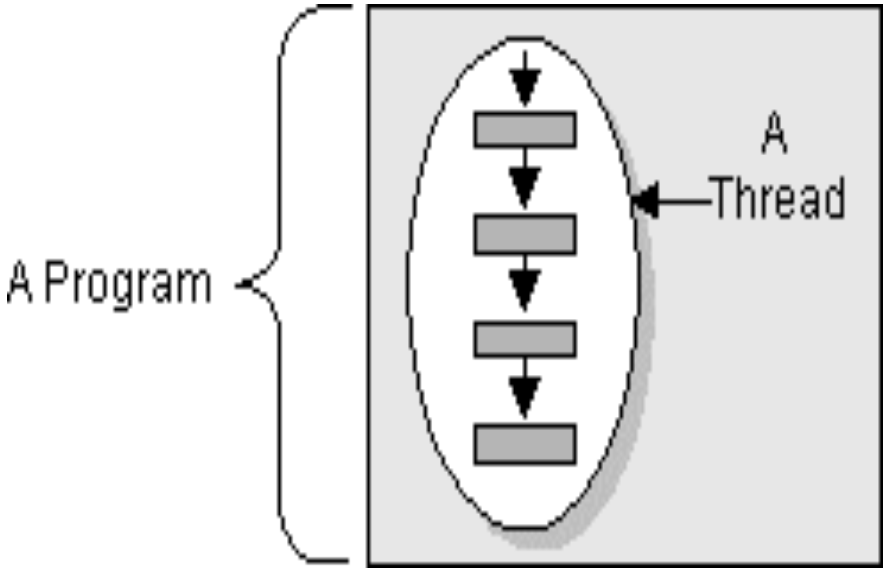
- 单线程的控制流程从main入口，直至main出口结束，中间经过一系列对象之间的相互协作→单一流程
- 如果单一流程中有些处理活动本身应该是并发的，这样的处理效率常常不能满足要求→多个并发流程→多个线程
  - CPU与OS都提供了并行/并发支持
- 任意两个并发流程cf1, cf2
  - 绝对并发：完全没有依赖关系
  - 部分并发：在特定条件下交换信息（即同步），然后独立并发执行
- 识别并发流程的基本策略
  - 问题域中独立且并发活动的实体
  - 调度器、电梯、请求输入装置？

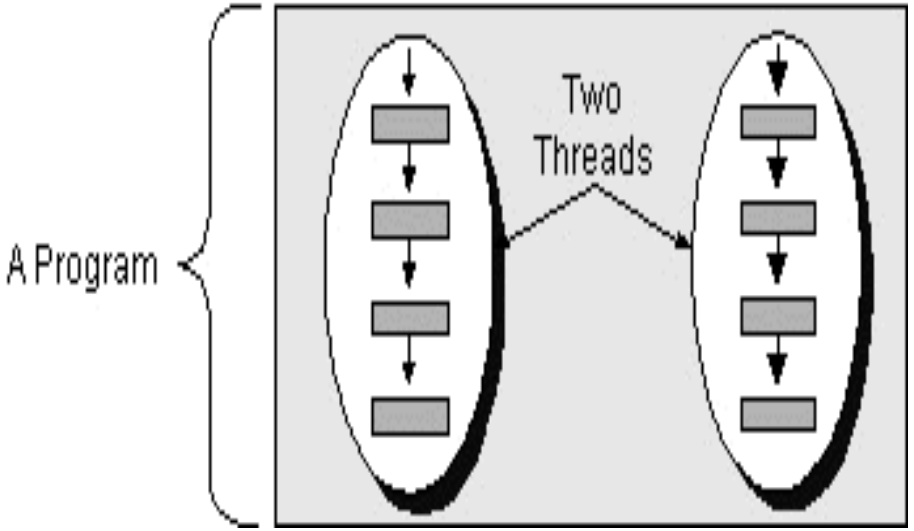
多线程处理

- 同步与异步
  - 同步：当一个方法执行返回时，调用者才能继续执行
  - 异步：不等方法执行返回，调用者继续执行
- 现实世界本质上是异步的
  - 每个对象同时在活动
  - 互相通知对方感兴趣的消息
  - 各自处理自己的消息
  - 在需要时进行同步

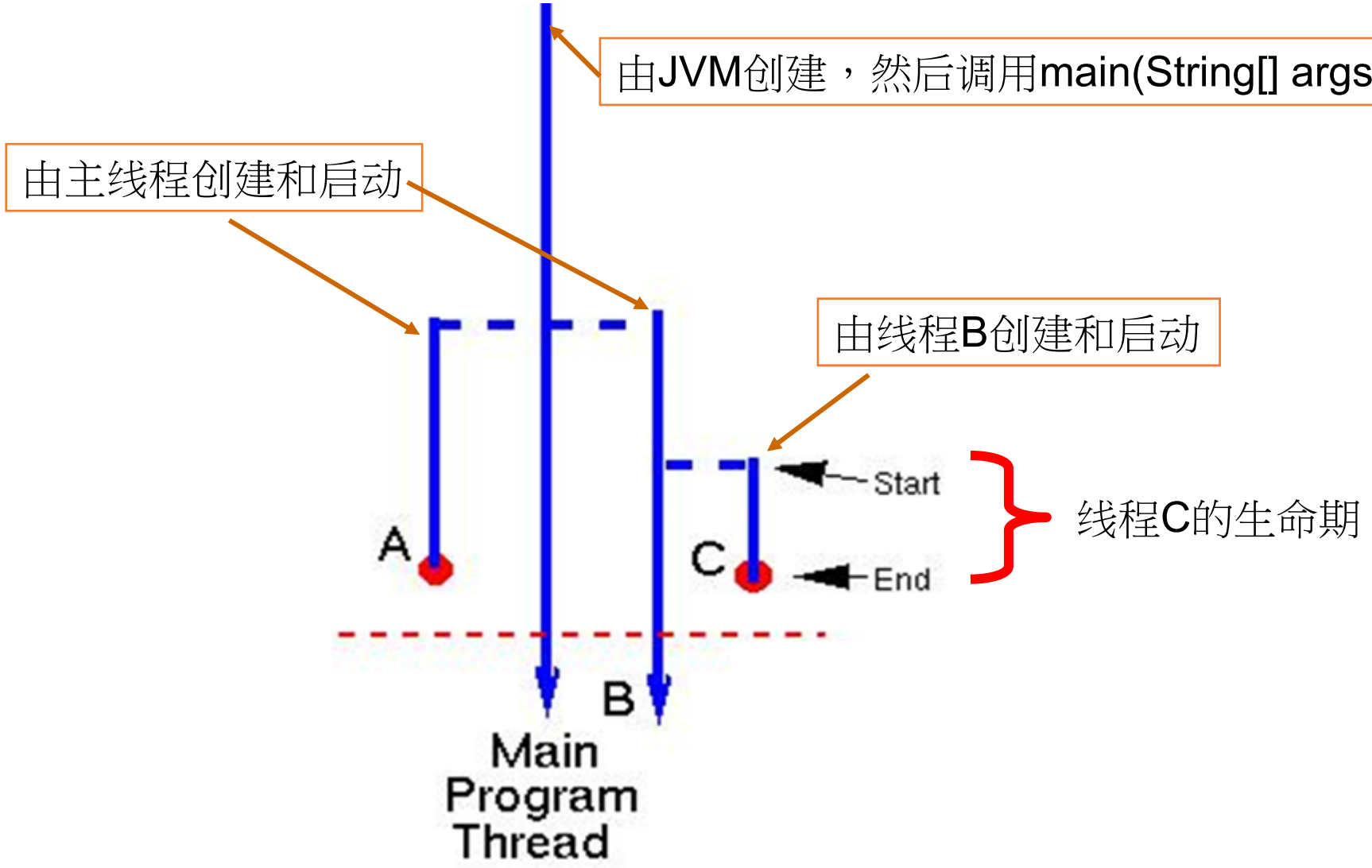
多线程处理

- 线程：程序内的一个顺序执行控制单位(流)(flow of control)
- 多线程程序：执行时有多个执行流
- 单线程程序：执行时只有一个执行流





多线程Java程序


由JVM创建，然后调用main(String[] args)
由主线程创建和启动
由线程B创建和启动
Start
End
线程C的生命期
A
C
B
Main
Program
Thread

多线程Java程序

- Java语言提供了对象化线程支持
  - Thread类和Runnable接口
  - 在java.lang包中定义
- 继承Thread类
- 实现Runnable接口

```
// 通过继承Thread类
public class Scanner extends Thread {
        public void run() {  // 线程执行入口点，相当于java程序的main()
                this.go();
        }
}

// 通过实现Runnable接口
public class Scanner implements Runnable {
        public void run() {
                this.go();
        }
}
```

多线程Java程序

- Thread和Runnable都是一种类型定义
- 如何启动线程执行（如何启动Java程序执行）？
  - run不是给用户代码来调用(main也不是给用户来调用!)
  - Thread t=new Scanner(…); // new Scanner (“1”);
  - t.start();

  - Thread t = new Thread(new Scanner(…)); //new Thread(new Scanner(…),”2”);
  - t.start();

能够直接调用对象t中的其他方法吗？会产生什么效果？

Thread的入口代码模板

```
public void run() {
 try {   …
       while (more work to do) {      // 常规的唤醒(即sleep()退出)从这里继续执行
              do some work;
              sleep( … ) or wait(...);          // 让其他线程有机会执行
            }
  }
  catch (InterruptedException e) { // 如果由interrupt()唤醒则从这里继续执行
      …  // thread interrupted during sleep or wait
  }
}
```

带有不确定性的线程调度

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

共享资源的访问控制

- 多个线程共同访问共享资源
  - 读写共同的对象
  - 如果不加以控制会导致数据状态混乱---数据竞争、数据不一致
    - 多个线程对变量的访问次序无法预测

Thread t1

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

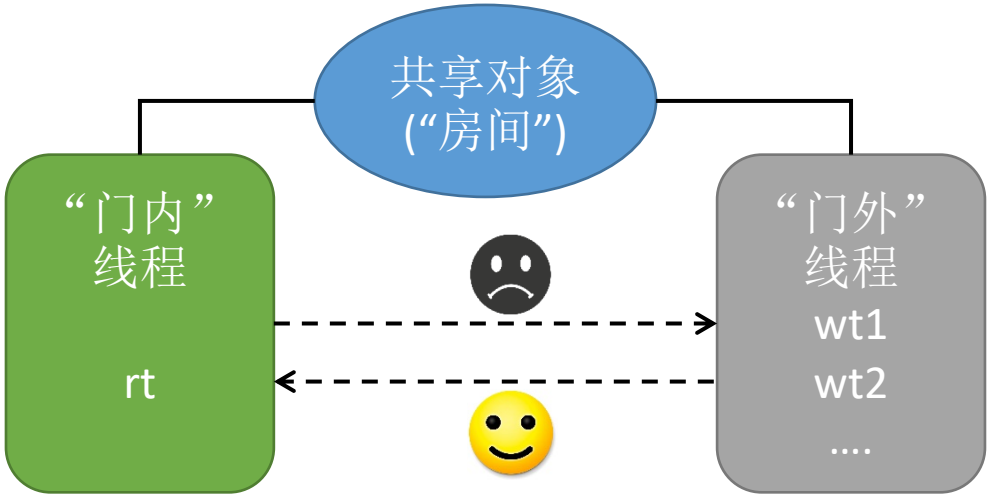
Thread t2

```
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？

共享数据的访问控制

- 采用互斥控制：任何时刻只允许一个线程获得访问/执行权限
  - synchronized(obj) {…} ：任意时刻只允许一个线程对对象obj进行操作
  - **synchronized method(…){…} 任意时刻只允许一个线程调用方法method**
- 任何线程访问受控的共享数据时
  - 可能有多个其他线程在等待访问该共享资源
  - 执行结束前通过notify/notifyAll来让JVM调度等待队列中的线程来访问共享数据
  - notify和notifyAll是否有区别？



线程交互

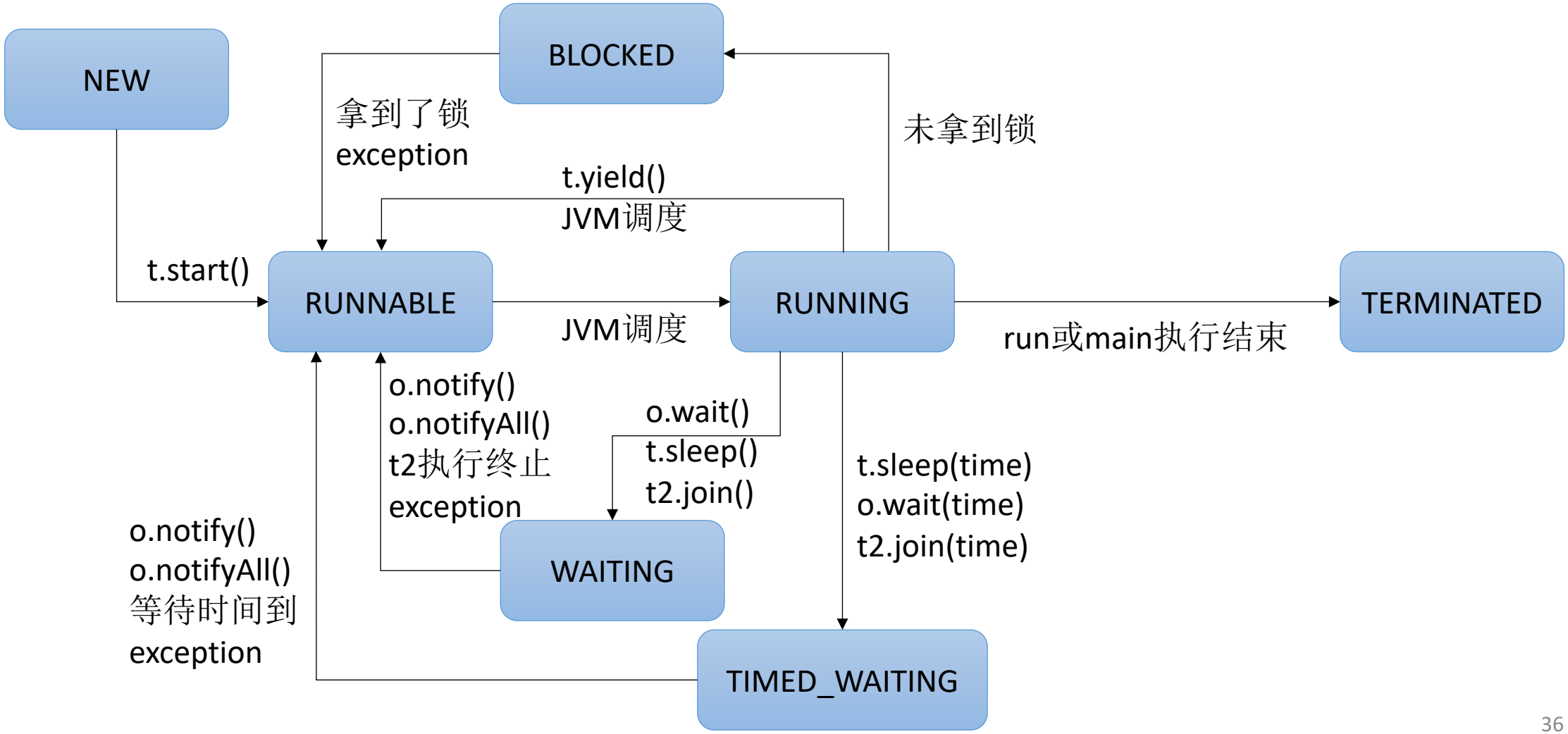
- 多线程程序与单线程程序的重要差异
  - 线程之间不具有调用关系，但可以使用Thread类或Runnable接口定义的调度操作来进行交互
  - 调试不能采用断点方式
- 线程之间不可避免会交互
  - 调度交互、数据交互
- 调度交互
  - 直接调度交互：启动(start)、结束(stop)、睡眠(sleep)、暂停(yield)
  - 间接调度交互：通过共享对象，wait, notify, notifyAll
- 数据交互
  - 通过共享对象交换数据

线程交互行为与线程状态紧密相关

Thread对象的状态

NEW, // 线程对象被创建后的初始状态
RUNNABLE,
// start()后所处状态: 正运行(running)或准备被调度(ready)
BLOCKED, // 阻塞状态，无法获得公共数据访问或临界区执行权限
WAITING, // 等待被唤醒状态，没有时限: wait(), join()
TIMED_WAITING, //等待指定时间: sleep(time), wait(time), join(time)
TERMINATED/DEAD // run()执行结束或stop()被调用

Thread的状态变化机制



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
- 通过构造器获得与其他线程共享的对象
  - 数据交互窗口
- 创建和使用专属对象
  - 仅供自己这个线程使用
  - 这些对象之间仍然可以相互调用方法
- 通过共享对象与其他对象交互
  - 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

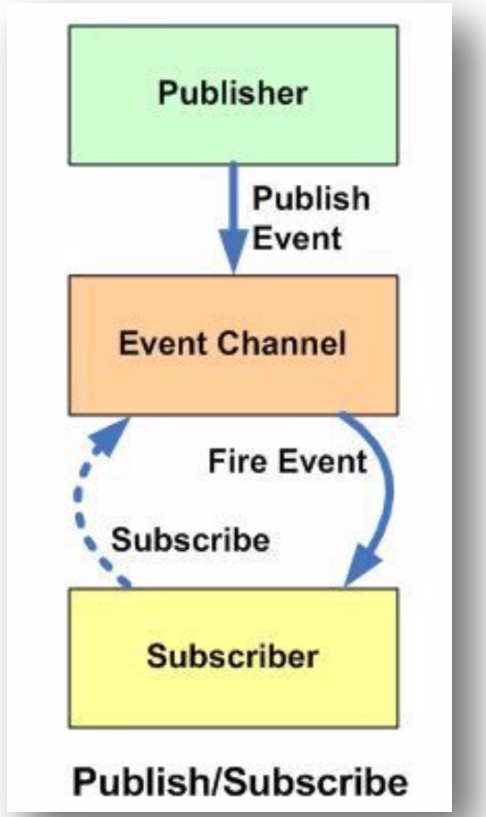
单线程程序与多线程程序的对比

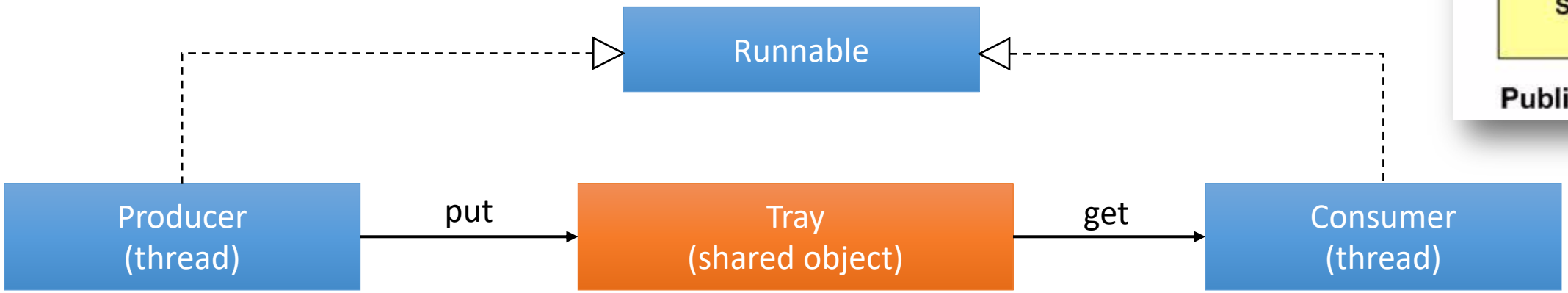
单线程下线程处理流程	多线程下线程处理流程
一个线程处理一个流程	一个线程处理一个流程
整个程序只有一个流程在执行	整个程序可能有多个流程在执行
执行不会被中断	执行会被中断
访问对象时无需额外保护	需要使用锁来保护对共享对象的访问

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，无需额外保护	相互间可以访问，但需用锁来保护
不具备共享控制锁的效果	具备共享控制锁的效果
可以访问共享对象，但需用锁来保护	不可以访问非共享对象

多线程交互模式：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
- 又可称为发布订阅模式





多线程交互模式：生产者消费者

```
public class Producer extends Thread {
    private Tray tray;          private int id;
    public Producer(Tray t, int id) {
    tray = t;          this.id = id;          }
    public void run() {
        int value;
        for (int i = 0; i < 10; i++)
         for(int j =0; j < 10; j++ ) {
            value = i*10+j;
            tray.put(value);
            System.out.println("Producer #" + this.id   + " put: ("+value+ ").");
            try { sleep((int)(Math.random() * 100));  }
            catch (InterruptedException e) { }
         };
     }
}
```

多线程交互模式：生产者消费者

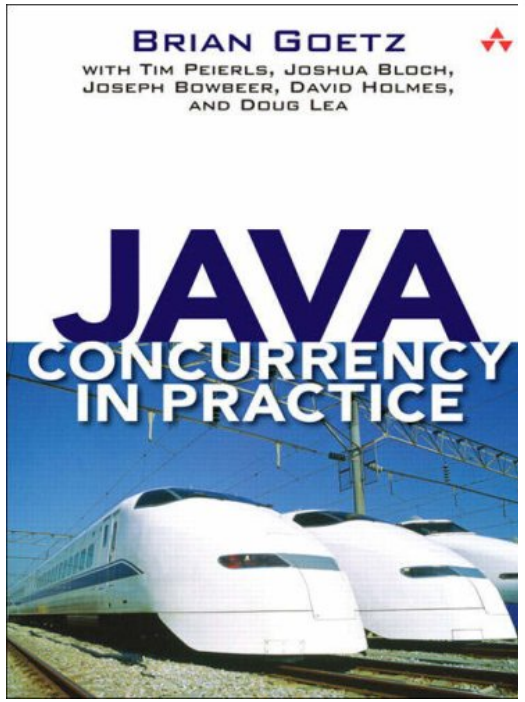
```
public class Consumer extends Thread {
    private Tray tray;
    private int id;
    public Consumer(Tray t, int id) {
        tray = t;          this.id = id;    }
    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = tray.get();
            System.out.println("Consumer #" + this.id + " got: " + value);
        }
    } }
```

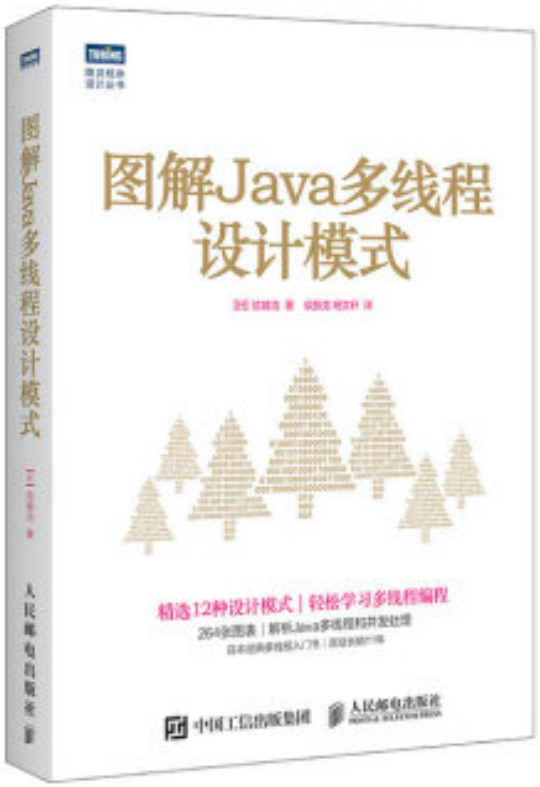
多线程交互模式：生产者消费者

```
public class Tray {
   private int value;     private boolean full = false;
   public synchronized int get() {
      while (full == false) {
         try {  wait();      } catch (InterruptedException e) { }       }
      full = false;              // 此时full为true，设为false后确保所有其他消费者不可能来抢
      notifyAll();
      return value;
   }
   public synchronized void put(int v) {
      while (full == true) {
         try {  wait();   } catch (InterruptedException e) { }       }
      full = true;         // 此时full为false，设为true后确保所有其他生产者不可能来抢
      value = v;
      notifyAll();
   }
}
```

多线程交互模式：生产者消费者

```
public class ProducerConsumerTest {
    public static void main(String[] args) {
        Tray t = new Tray();
        Producer p1 = new Producer(t, 1);
        Consumer c1 = new Consumer(t, 2);
        p1.start();
        c1.start();
    }
}
```





- 生产者和消费者只是线程的角色
- 可以构造生产者-消费者链，实现pipeline结构
- 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

进一步问题分析与实现

- 假设消费者不知道生产者会生产多少货物怎么办？
  - 只要有新的货物，就去消费
- 如果消费者的消费速度和生产速度匹配不上怎么办？
  - 需要使用队列来管理货物（缓冲区策略）
- 如果没有生产者怎么办？
  - 事先给消费者准备好了货物，消费完就ga

```
public class Tray {
    private ArrayList<In
    private int limit;
    public Tray(int vol)
ArrayList<Integer>();l
    public synchronize
        return values.is
    }
```

```
public class Consumer extends Thread {
    private Tray tray;…
    public void run()
        int value;
        while(!tray.is
        value = tray.g
        System.out.p
    }
}
```

```
for(…)t.put(…);//备好货
Consumer c1 = new Consumer(t, 1);
Consumer c2 = new Consumer(t, 2);
Consumer c3 = new Consumer(t, 3);
c1.start(); c2.start();c3.start();
```

```
public  synchronized void put(int v){
        while(values.size()>=limit){
        try{wait();}catch (InterruptedException e){…}
        d(new Integer(v));
        );
```

```
        zed int get(){
        ize()==0){
        atch (InterruptedException e){…}
        emove(values.size()-1).intValue();
        return v;
    }
```

多线程交互模式：单例模式

- 有的系统需要一个全局对象进行协调，比如系统配置文件，全局只有一个，电梯系统的调度器，全局只有一个（电梯作业都只有一个调度器）
- 单例模式是一种非常有效的全局对象保存和访问方法
  - 避免把电梯调度器这个对象到处传递

基本要求

希望对象只创建一个实例，并且提供一个全局的访问点。

设计思路

1、将构造方法定义为Private，这样其他处的代码就无法通过调用该类的构造方法来实例化该类的对象，只有通过该类提供的静态方法来得到该类的唯一实例；

2、提供一个静态方法，当调用这个方法时，如果类持有的全局对象引用不为空就返回这个引用，否则就创建实例，并保存好。

多线程交互模式：单例模式

```
public class Singleton {
    private final static Singleton INSTANCE = new Singleton();
    private Singleton(){...}
    public static Singleton getInstance(){
        return INSTANCE;
    }
}
```

- 优点
  - 类装载时完成INSTANCE实例化，此后所有线程访问的INSTANCE对象都是一开始实例化的那个对象
- 缺点
  - 浪费内存

- 构造方法private：外部无法直接通过new来创建新的实例（否则不受控）
- getInstance为static：因为不能通过new创建，必须提供static方法
- INSTANCE变量private：不允许通过Singleton.INSTATNCE访问
- INSTANCE变量final：不允许对INSTANCE对象进行修改
- INSTANCE变量static：static方法不能访问非static变量

多线程交互模式：单例模式

```
public class Singleton {

    private static Singleton singleton;

    private Singleton() {}

    public static Singleton getInstance() {
        if (singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

- 假设一个线程正好在判断if(singleton==null)完成这个语句，准备执行new Singleton之前，另一个线程也执行了if(singleton==null)语句，两个线程都执行new Singleton方法，最后创建了两个singleton实例

多线程交互模式：单例模式

```
public class Singleton {

    private static Singleton singleton;

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (singleton == null) {
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

- 每个线程在想获得类的实例时候，执行getInstance()方法都要进行同步。
- 效率太差

多线程交互模式：单例模式

```
public class Singleton {

   private static Singleton singleton;

   private Singleton() {}

   public static Singleton getInstance() {
      if (singleton == null) {
         synchronized (Singleton.class) {
            singleton = new Singleton();
         }
      }
      return singleton;
   }
}
```

- 通过Singleton.class对象加锁
  - Java.lang.Class的对象
- 线程安全了吗？
- singleton==null没有保护起来

多线程交互模式：单例模式

```
public class Singleton {

    private static volatile Singleton singleton;

    private Singleton() {}

    public static Singleton getInstance() {
        if (singleton == null) {
            synchronized (Singleton.class) {
                if (singleton == null) {
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

- 通过两次singleton==null进行检查，这样最差的结果是两个线程进行了同步，一旦创建以后，所有线程在第一次singleton==null处不再需要进行同步判断了

作业

- 多线程电梯系统
  - 一部电梯
  - 输入和输出由我们提供了，再也不用担心WF的问题啦
- 电梯运动会消耗时间
  - 上升或下降一层的时间：0.5s
  - 电梯开关门的时间：0.25s
- 请求定义
  - id-FROM-x-to-y：乘客从x到y（我们假设这里的电梯提供这样的输入装置）
  - 你需要从中解析出相应的运动方向要求
- 调度策略
  - 自行定义，但必须响应所有的合法请求
  - 评测机采用的模拟机采用FAFS策略，得到的调度时间作为基准

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
  - 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
- 调度器对象是全局唯一，是否可以使用单例模式进行获取？
- 乘客请求怎么存？ArrayList/HashMap？访问这个容器是否需要同步？
- 电梯是否需要存储需要响应的请求？如何存储？如何访问？