

# 第五讲

# Java对象运行机制与多线程

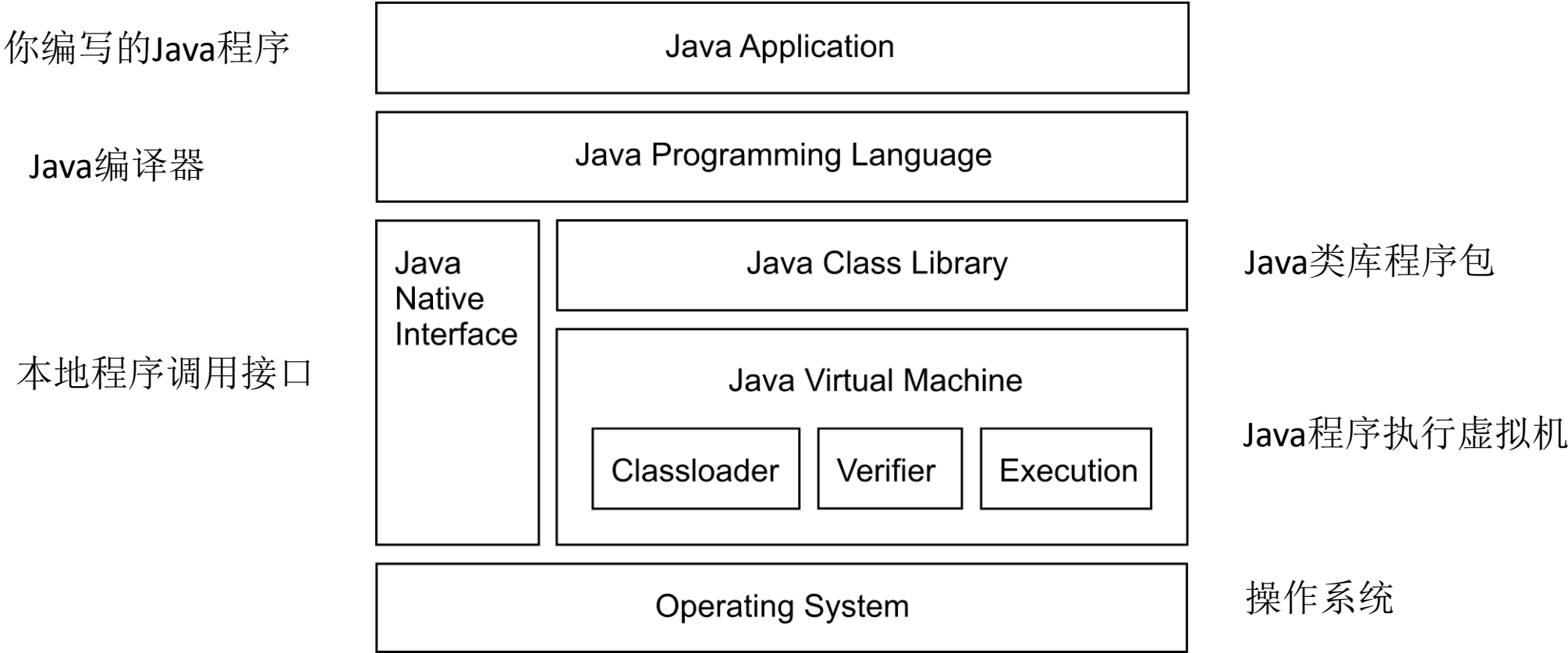
OO课程组2018

北京航空航天大学计算机学院

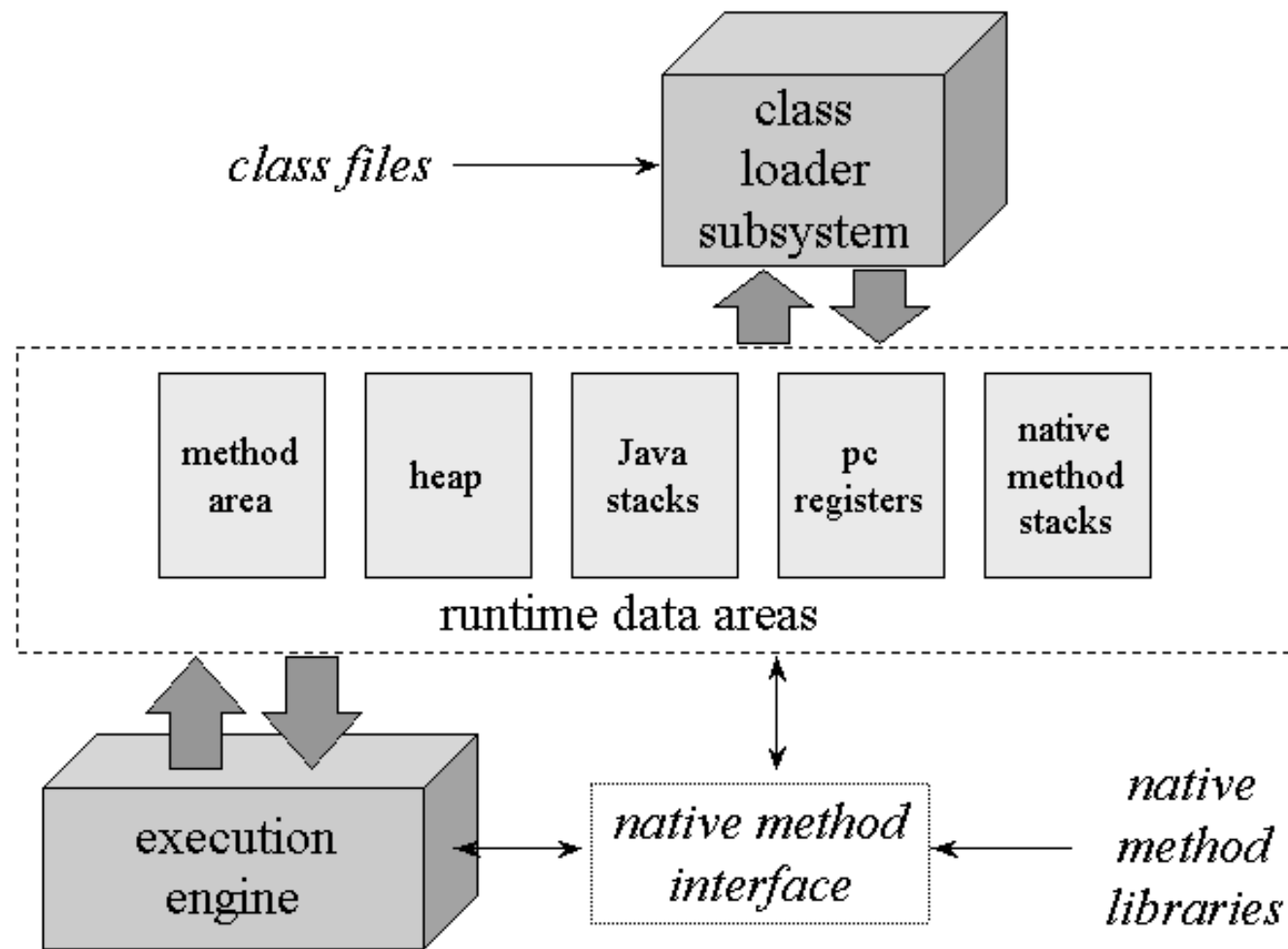
# 内容摘要

- Java系统概览
- Java应用如何执行
  - JVM基本结构
  - 类加载
  - 对象初始化
  - JVM内存划分
  - 对象方法调用
- Java应用的多线程处理初步介绍
- 作业

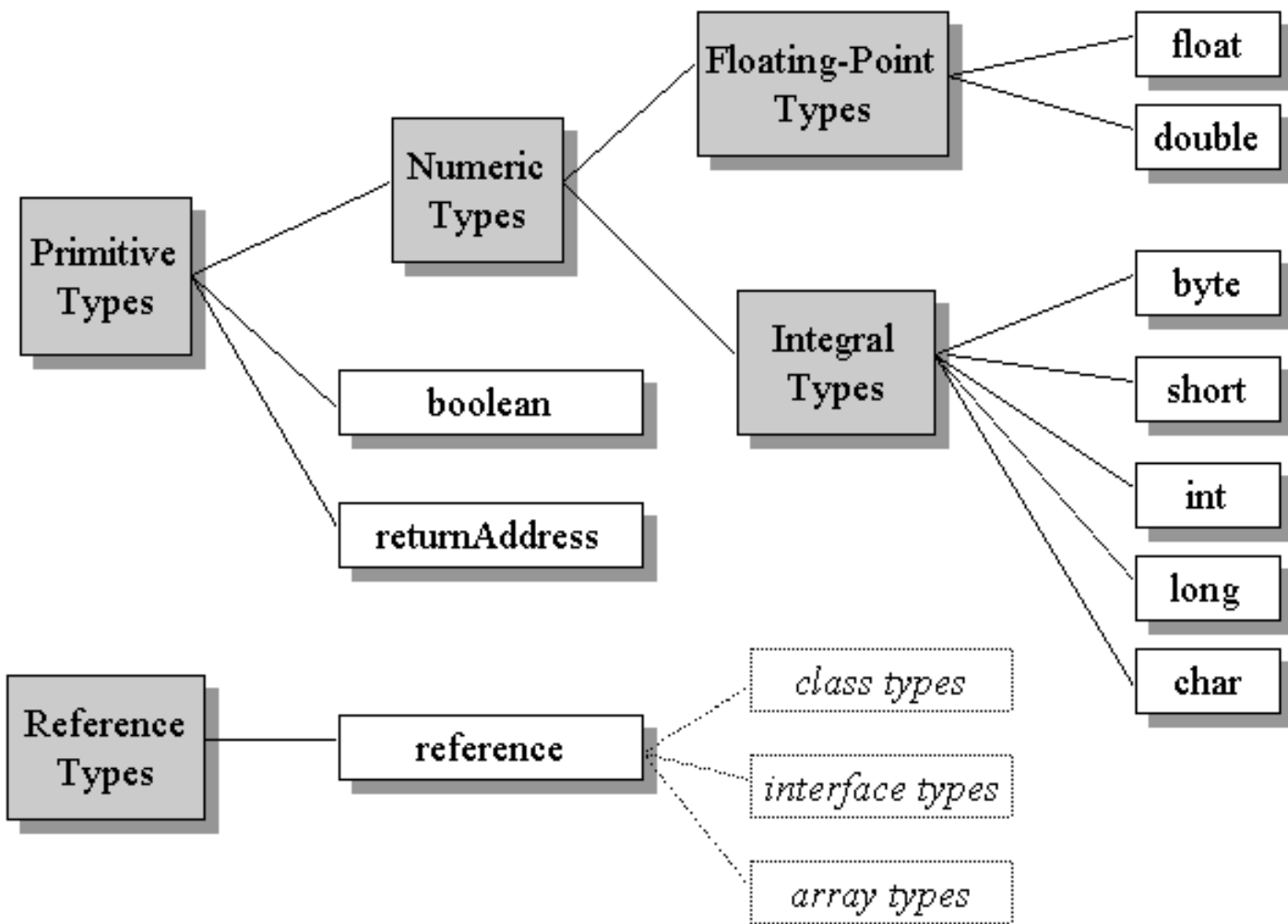
# Java系统概览



# Java虚拟机(Java Virtual Machine)



# JVM中用到的数据类型



# JVM中的classloader

- JVM在程序运行过程中根据需要动态加载相应的类
  - 从而能够按照程序的运行时行为所需来创建相应的对象
  - 把.class文件(可执行bytecode)加载进内存
- JVM的缺省loader可以被扩展
  - 扩展实现不同的加载策略
  - 扩展从不同的source加载class，如从网络加载

# 动态加载、链接与初始化

阶段	功能	结果
加载(load)	根据类的完整名称获得相应的class文件	<code>byte[] data</code>
链接(link)	<ul style="list-style-type: none"><li>- 验证: 检查class文件中的bytecode</li><li>- 准备: 申请和初始化字节码中的数据</li><li>- 解析: 把之前的名字引用改变为对象引用</li></ul>	<code>Class c, resolved</code>
初始化(init)	触发类的初始化代码	<code>Class c, initialized</code>

# 验证bytecode

- `.class`文件中的bytecode可能来自于多种不同编译器
  - 其中可能包含危险甚至恶意代码
- 检查bytecode的正确性
  - 每个指令都有有效的操作码
  - 每个分支指令都要转移到某个指令的开始处，而不是中间位置
  - 每个方法都必须拥有结构上正确的签名
  - 每个指令都必须遵守Java类型规则

- Load and store (e.g. `aload_0`, `istore`)
- Arithmetic and logic (e.g. `ladd`, `fcmpl`)
- Type conversion (e.g. `i2b`, `d2i`)
- Object creation and manipulation (`new`, `putfield`)

- Operand stack management (e.g. `swap`, `dup2`)
- Control transfer (e.g. `ifeq`, `goto`)
- Method invocation and return

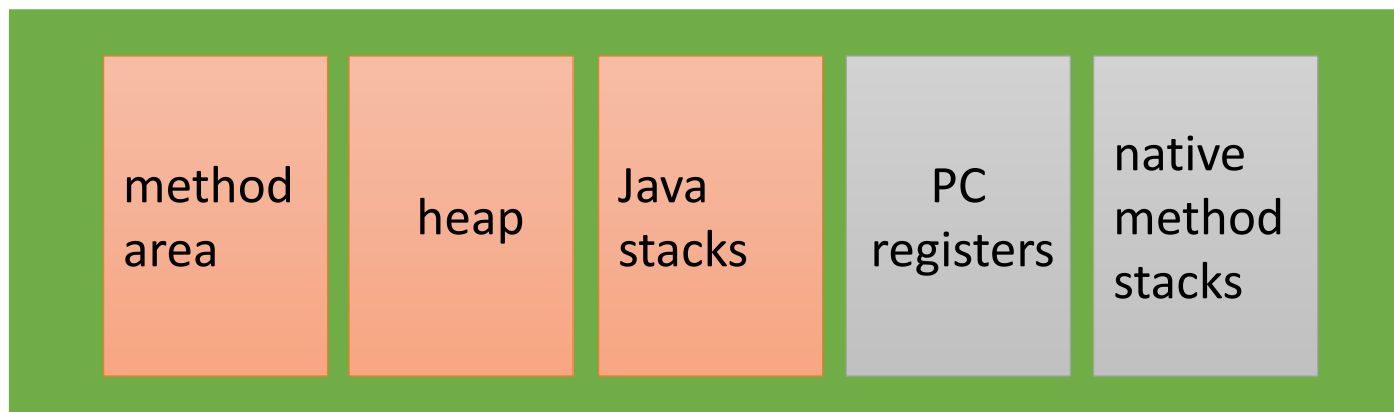


# JVM中的执行引擎

- 解释执行bytecode
  - 运行时检查，如数组越界检查，除零检查等
- 可以通过JNI接口调用本地方法
  - 典型如调用C语言编写的程序
- 多种手段来触发Java方法的调用
  - invokevirtual – 已知相应对象的类型
  - invokeinterface - 已知相应对象实现的接口
  - invokestatic – 触发静态方法的执行
  - invokespecial – 其他情况

# JVM内存区域划分

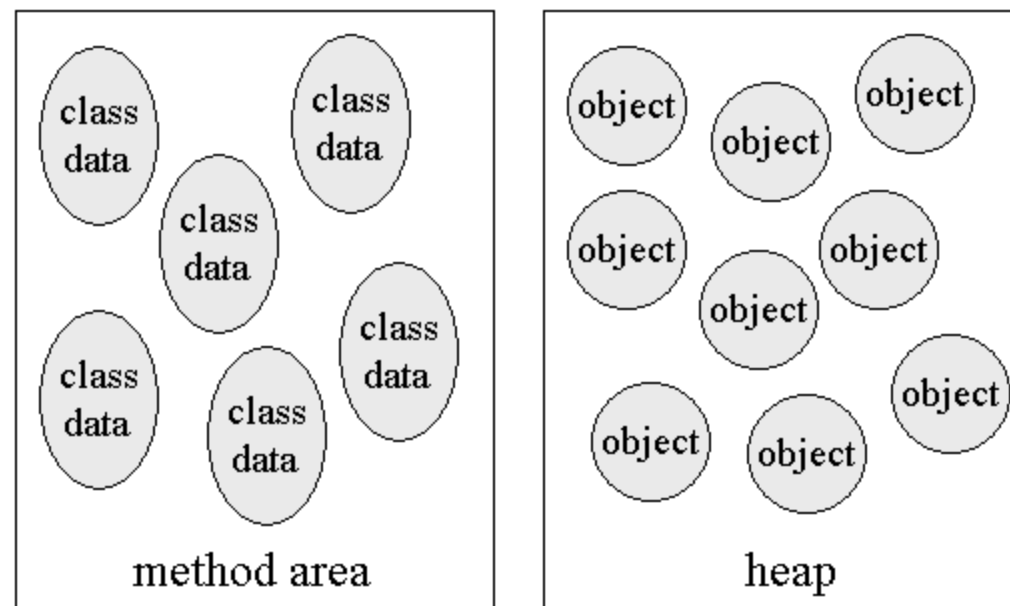
- Java程序运行时由一到多个线程组成
- 每个线程都有自己的栈
- 所有线程共享同一个堆



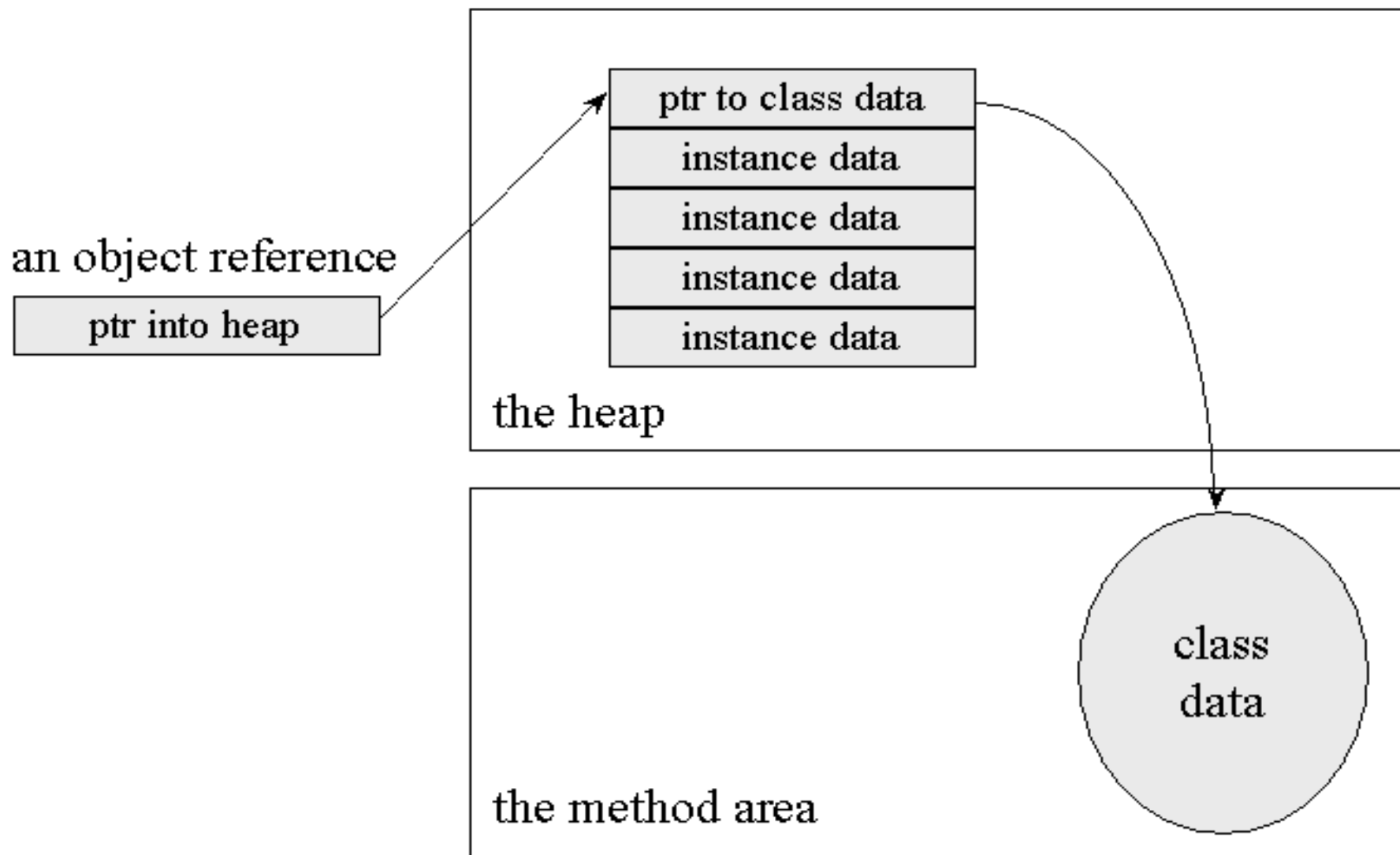
- Method area
  - Class description
  - Code
  - Constant pool
- Heap
  - Objects and Arrays
  - Shared by all threads
  - Garbage collection
- Stack
  - Invocation frame
  - Local variable area
  - Operand stack

# 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



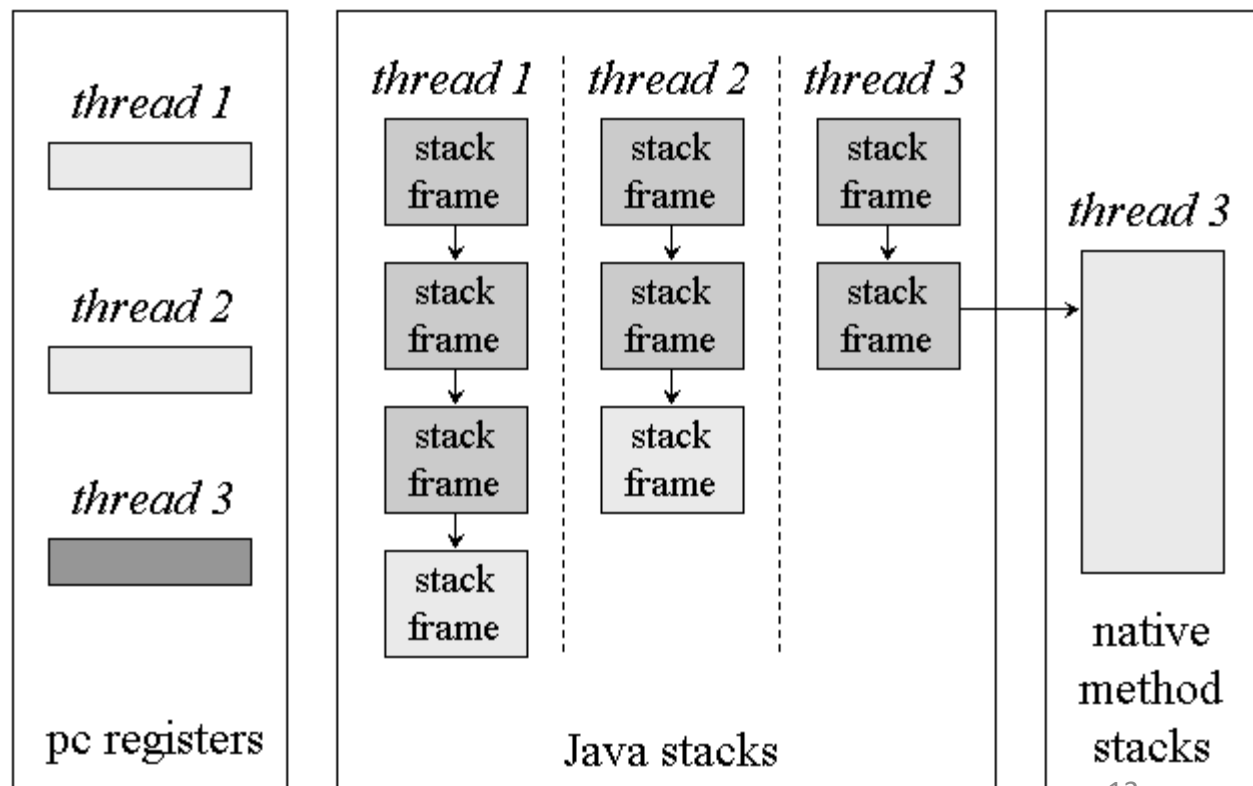
# 堆中的对象表示



# JVM中的内存划分

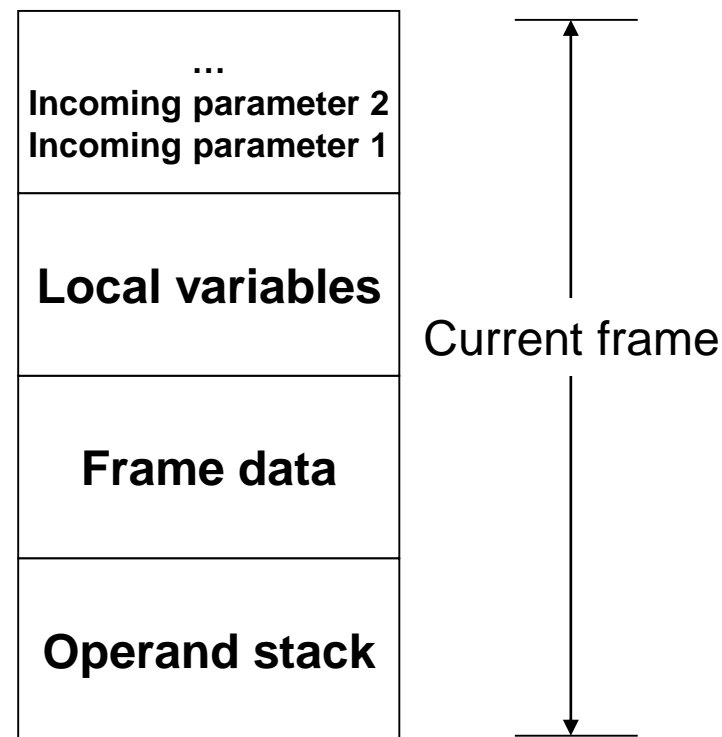
- 栈内存区

- 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



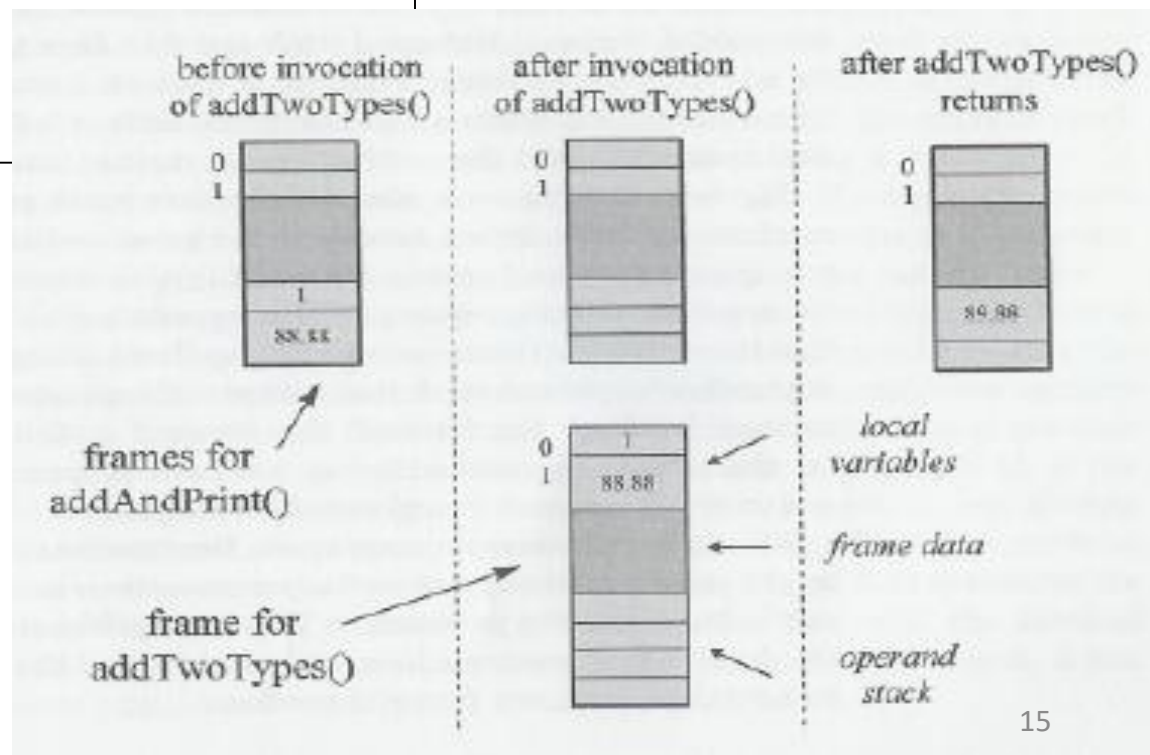
# 栈帧结构

- 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



# 方法调用时的栈帧变化

```
class StackFrameExample {  
    public static void addAndPrint() {  
        double result = addTwoTypes(1, 88.88);  
        System.out.println(result);  
    }  
  
    public static double addTwoTypes(int i, double d) {  
        return i + d;  
    }  
}
```



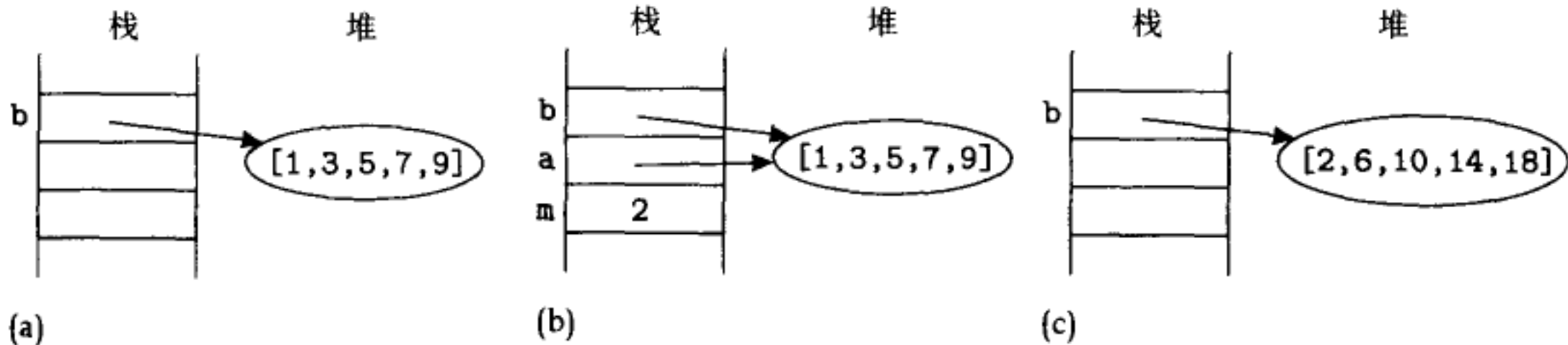
# 对象方法调用时发生了什么

- 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);
```

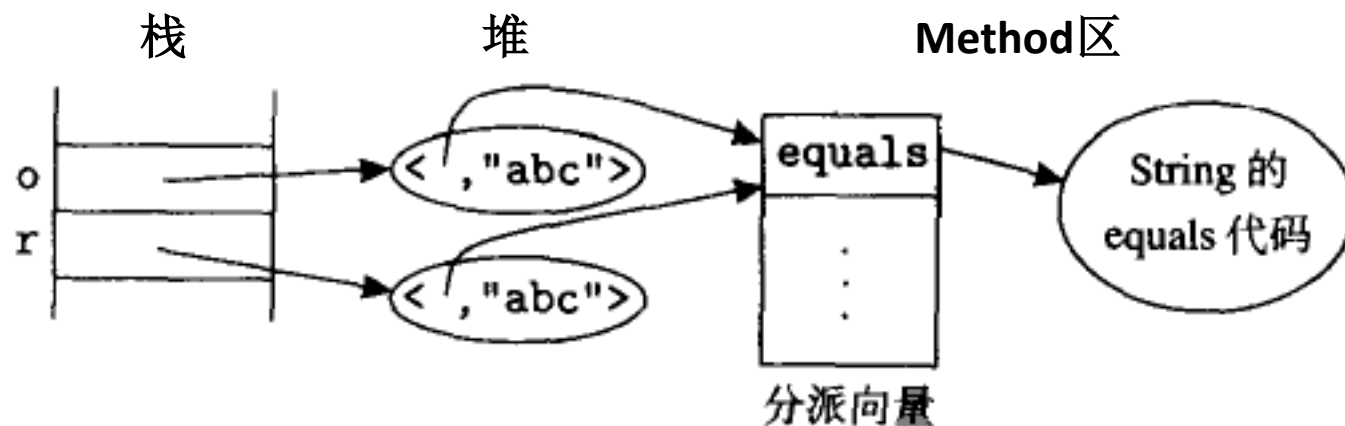




# 对象方法调用时发生了什么

- 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;  
    }  
}
```

要点

类的加载  
对象初始化  
方法调用

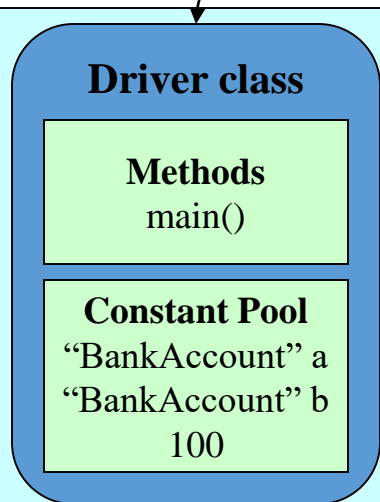
```
public class Driver {  
    public static void main( String[] args ) {  
        BankAccount a = new BankAccount();  
        BankAccount b = new BankAccount();  
        b.deposit( 100 );  
    }  
}
```

```
// In command prompt
java Driver

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

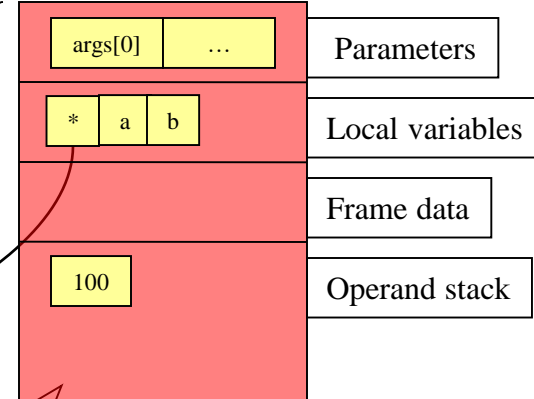
```
public class Driver {
    public static void main( String[] args ) {
        BankAccount a = new BankAccount();
        BankAccount b = new BankAccount();
        b.deposit( 100 );
    }
}
```

ClassLoader把  
Driver类加载到方  
法区域



**Method Area**

**main()**

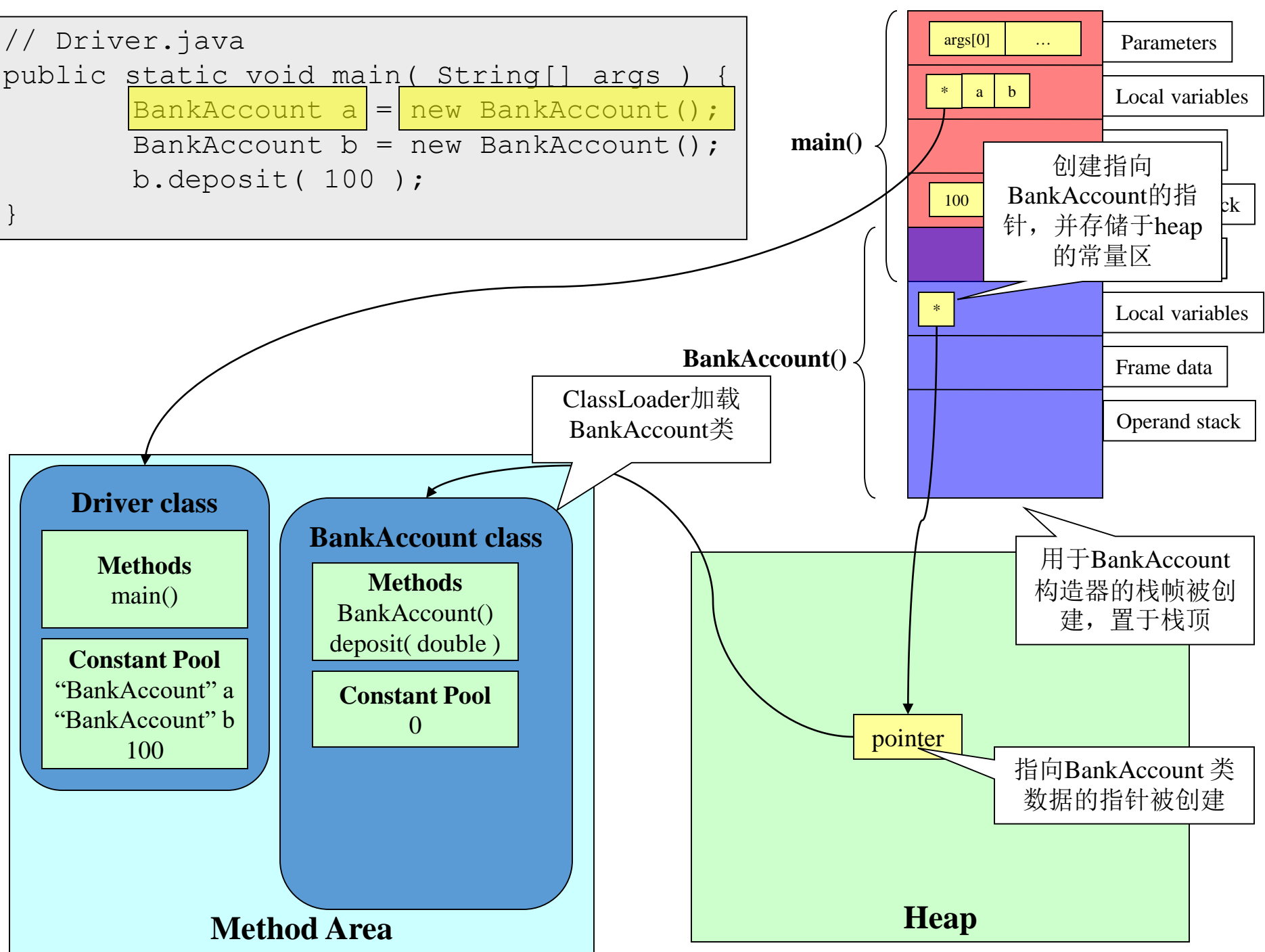


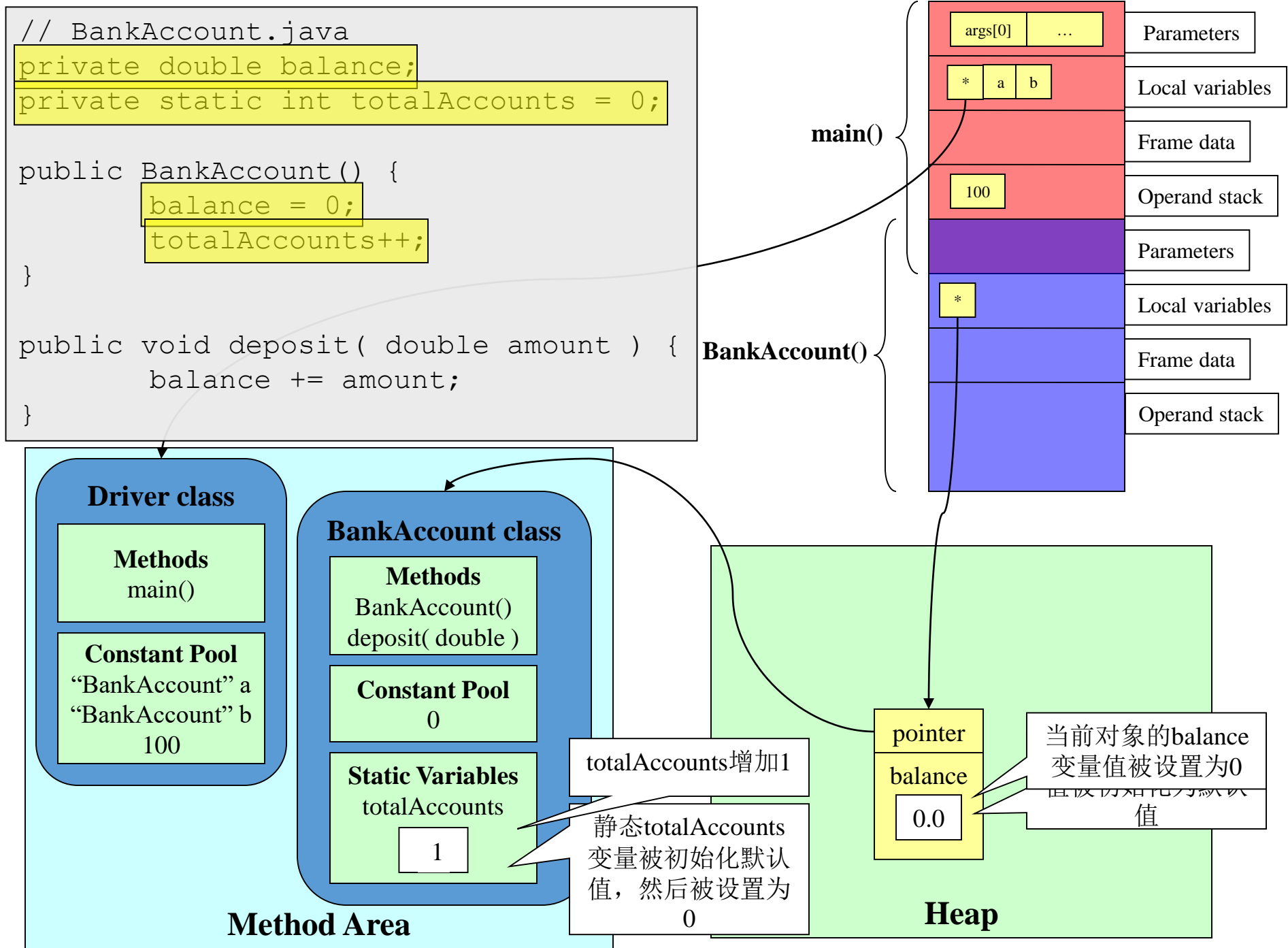
针对main()方法的栈帧  
被创建，并处于栈顶。  
\*为对当前对象对应  
class的引用。



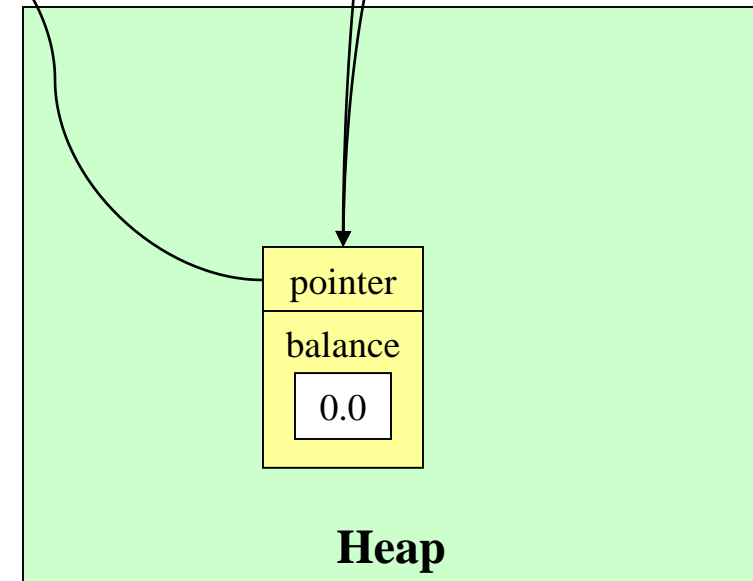
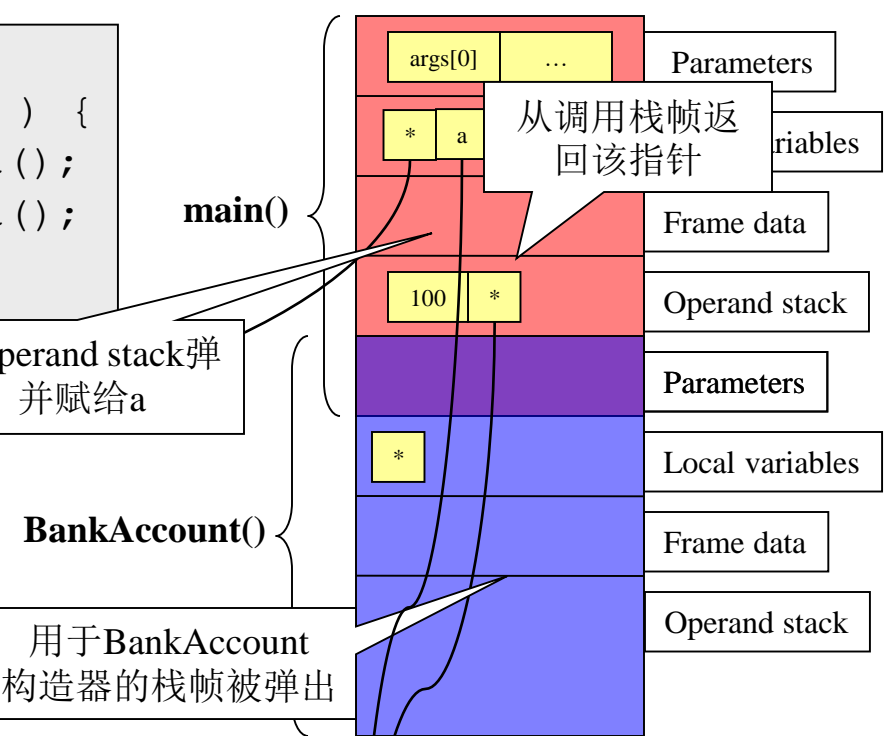
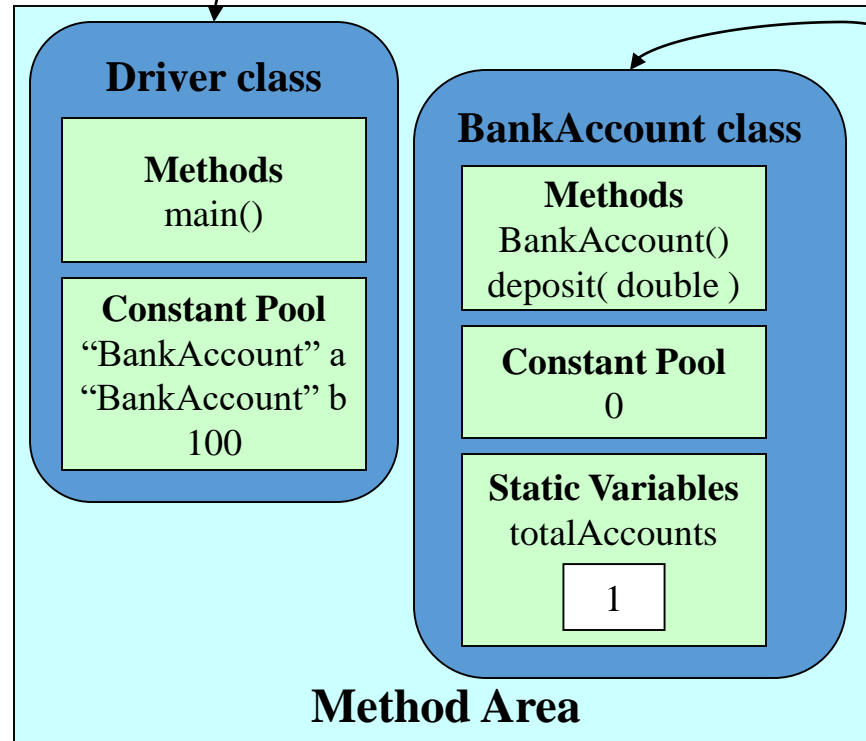
**Heap**

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





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

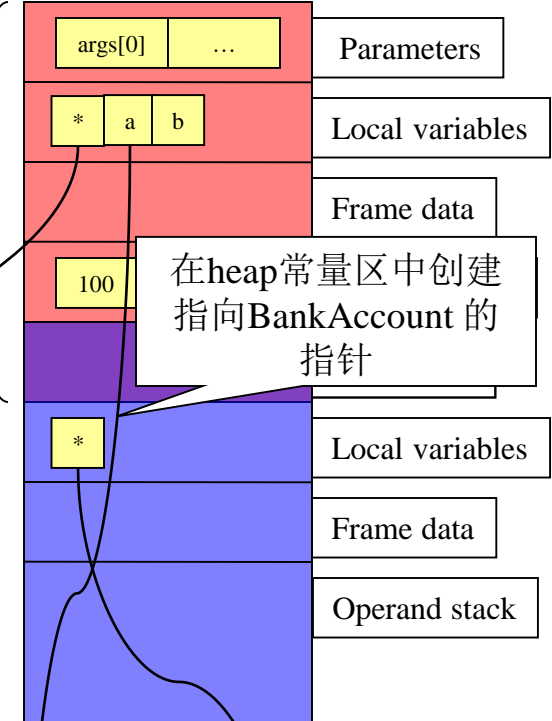


```
// Driver.java
public static void main( String[] args ) {
    BankAccount a = new BankAccount();
    BankAccount b = new BankAccount();
    b.deposit( 100 );
}
```

BankAccount类已经被加载，无需再次加载

main()

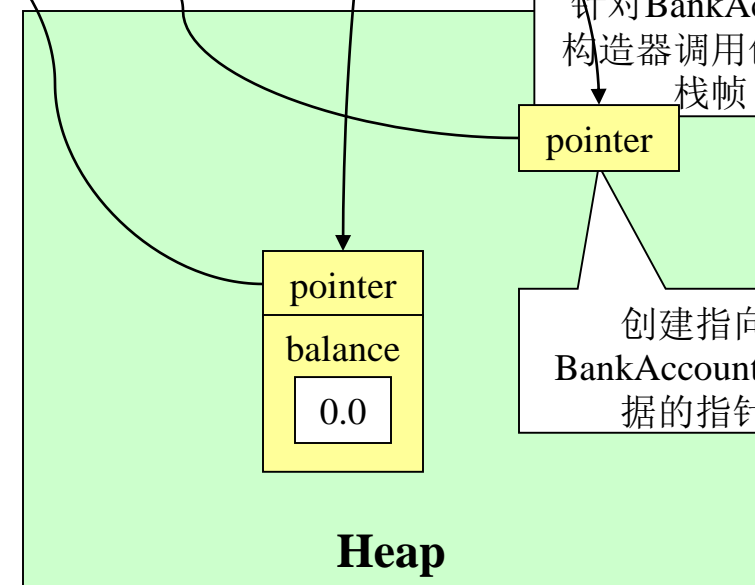
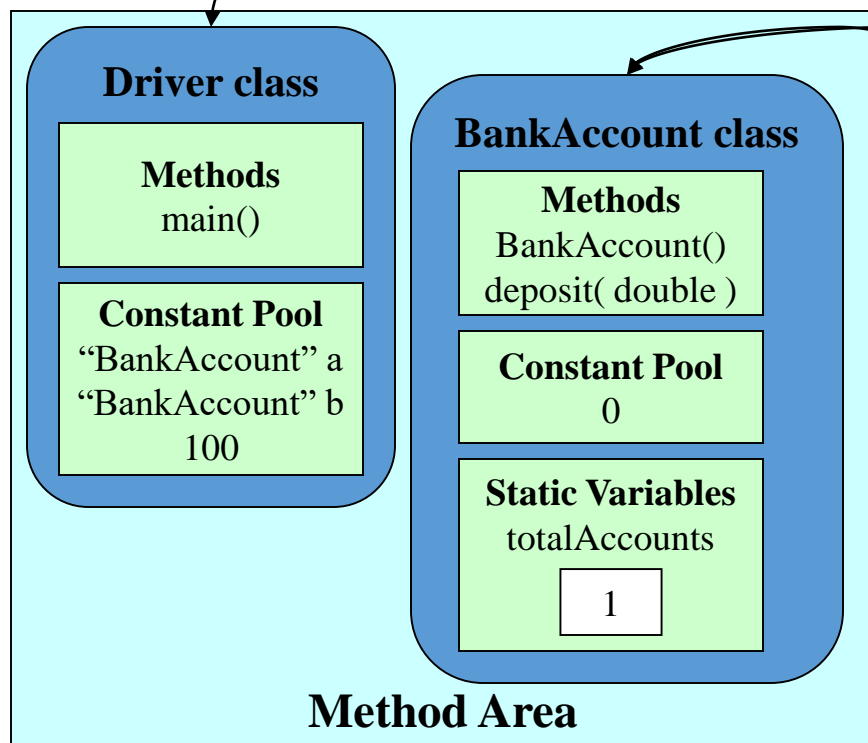
BankAccount()

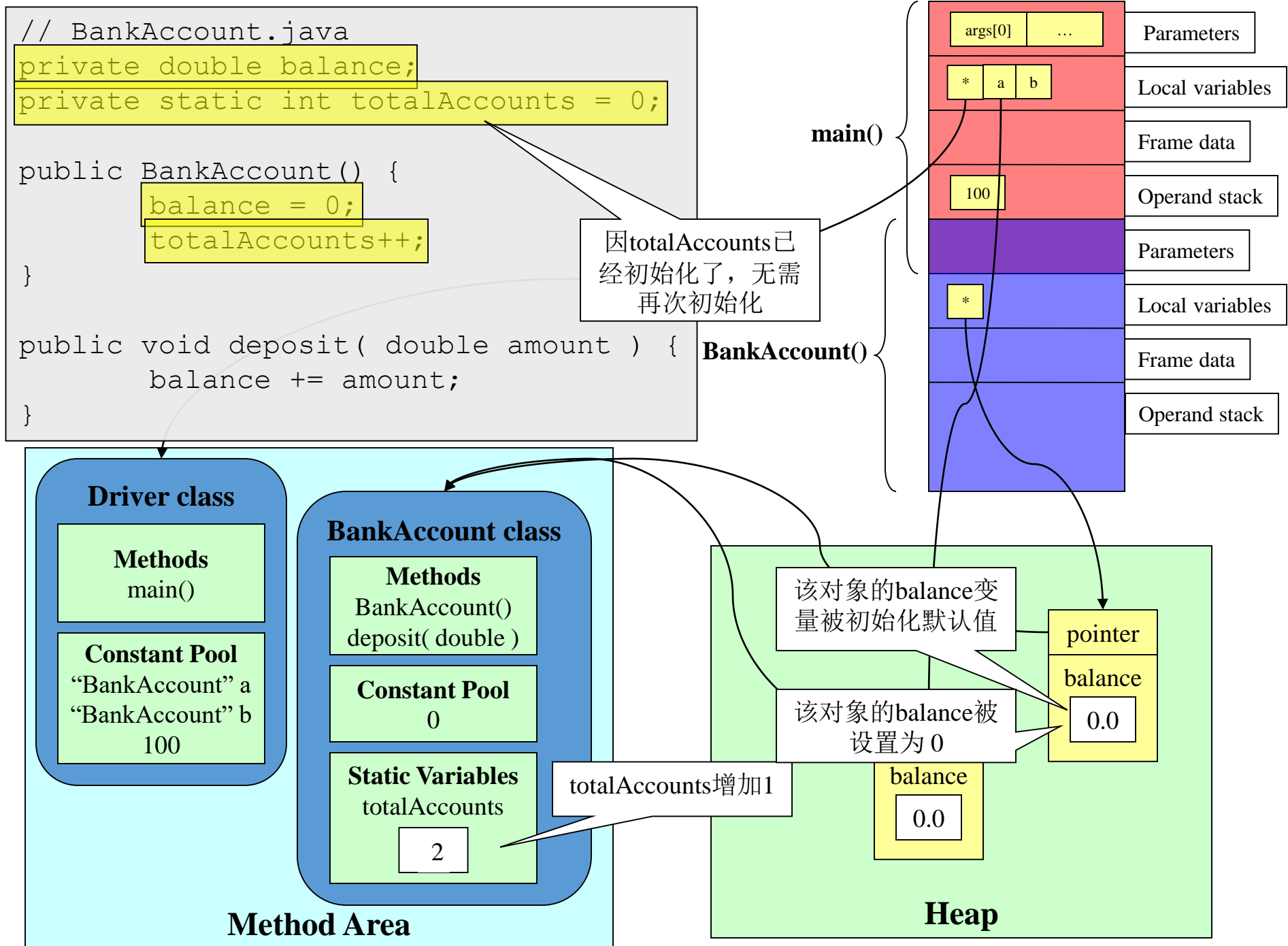


在heap常量区中创建指向BankAccount的指针

针对BankAccount构造器调用创建的栈帧

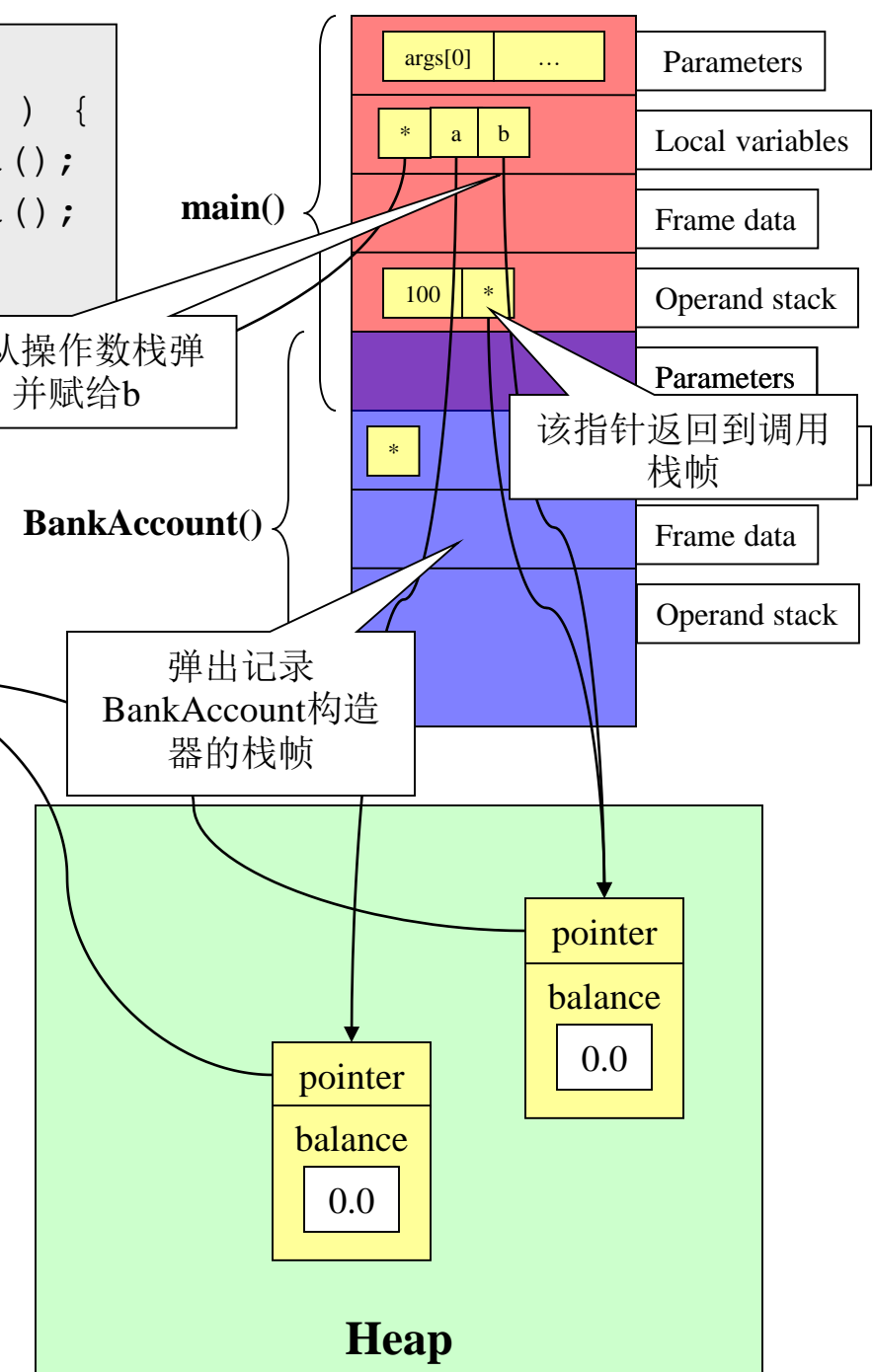
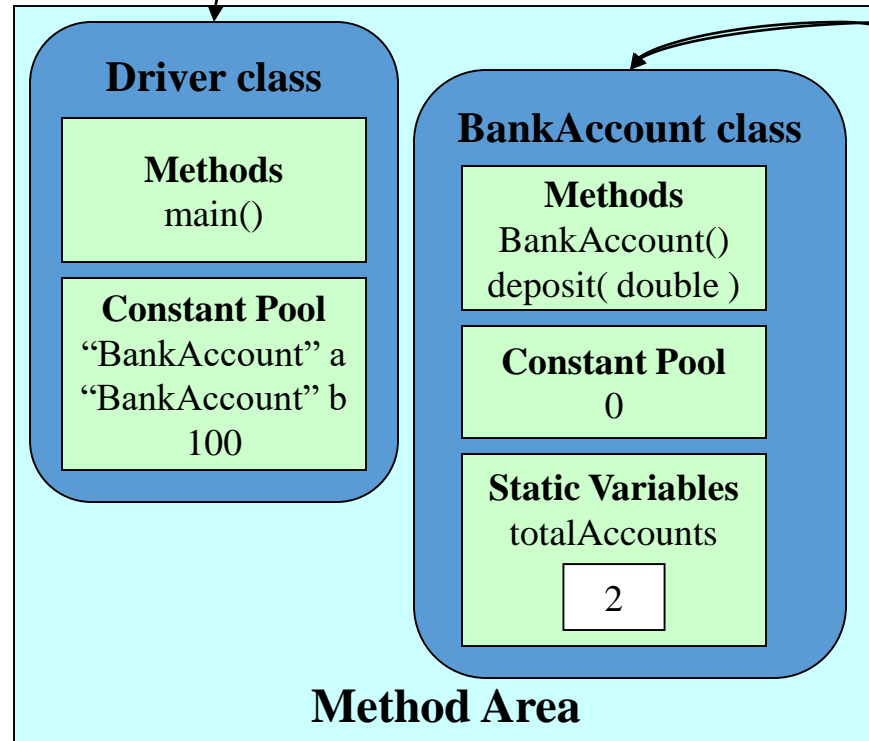
创建指向BankAccount类数据的指针







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

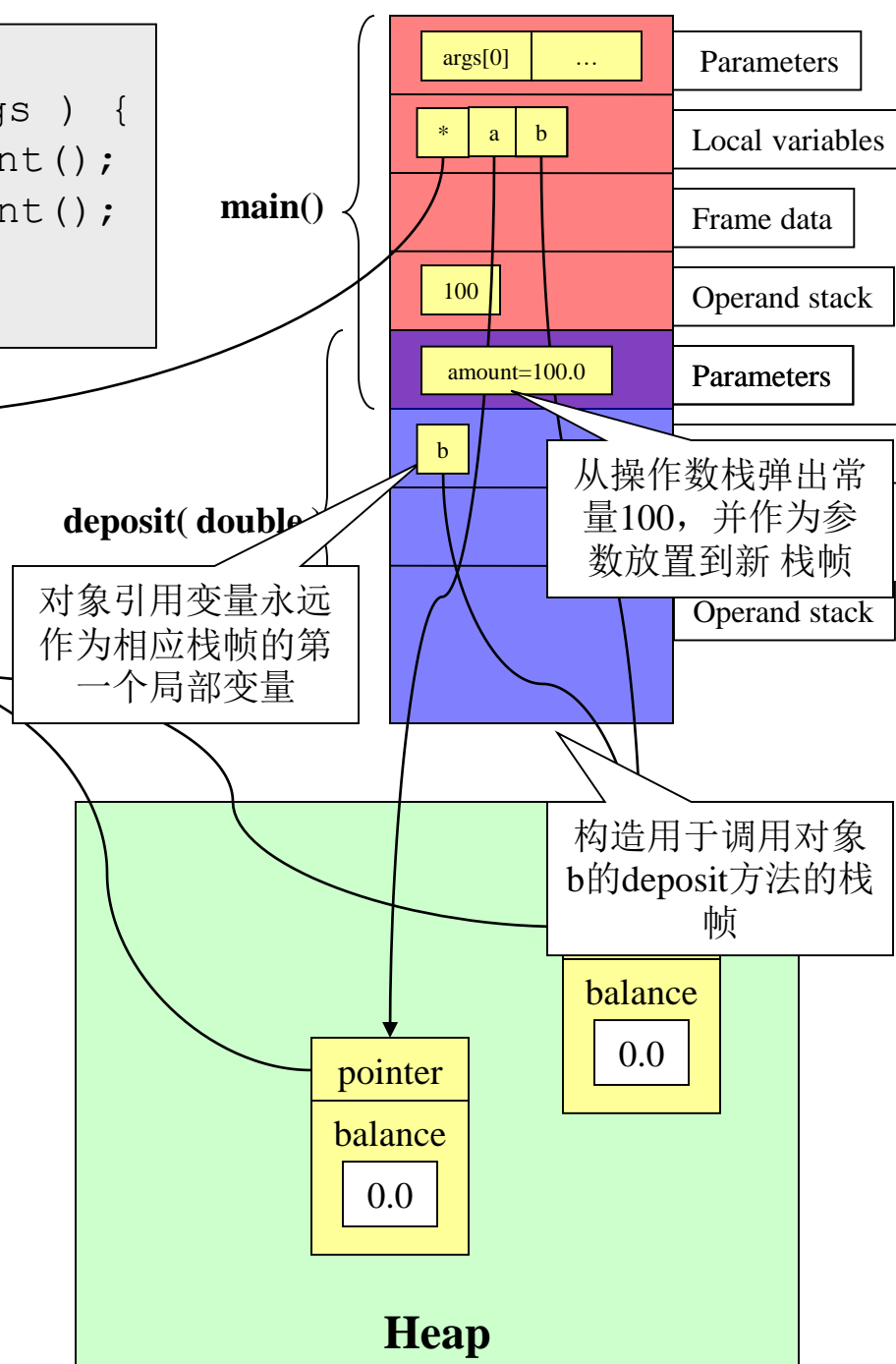
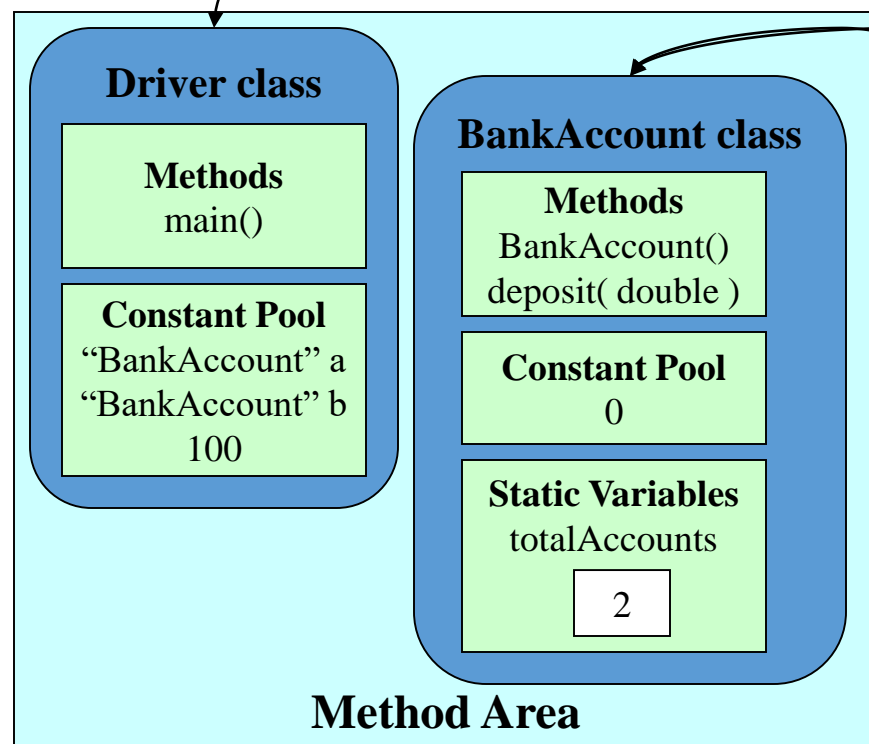


该指针从操作数栈弹出，并赋给b

该指针返回到调用栈帧

弹出记录 BankAccount构造器的栈帧

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



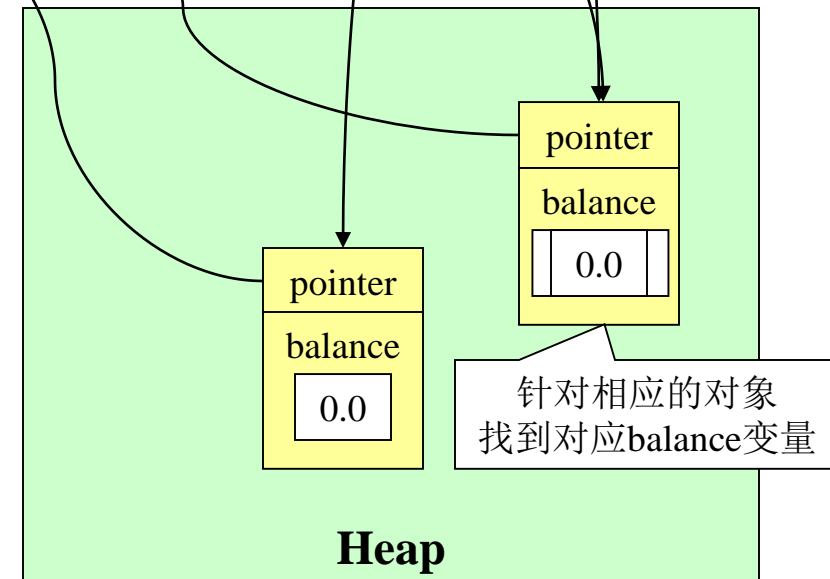
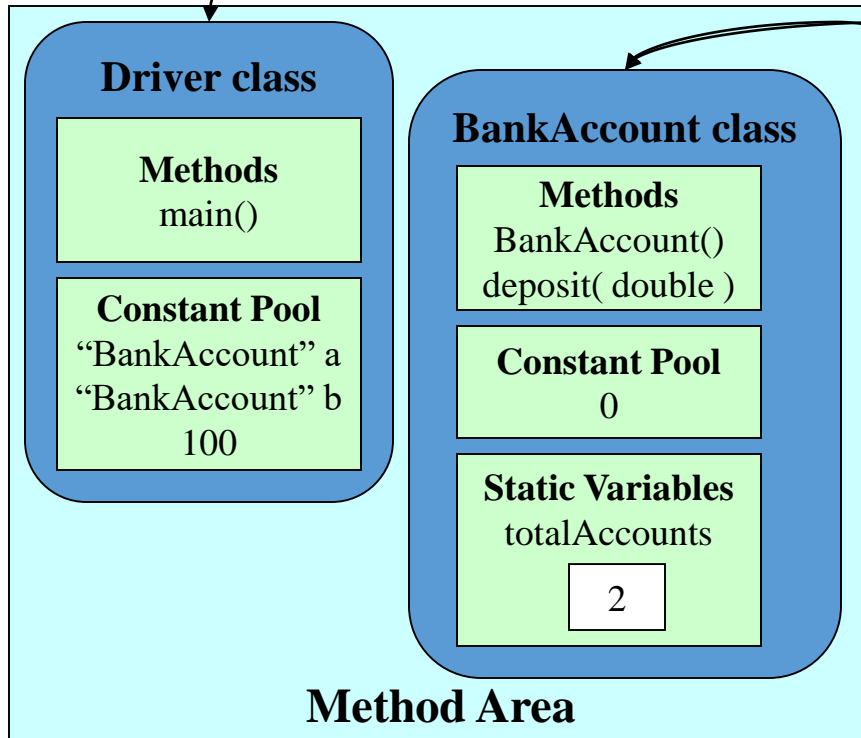
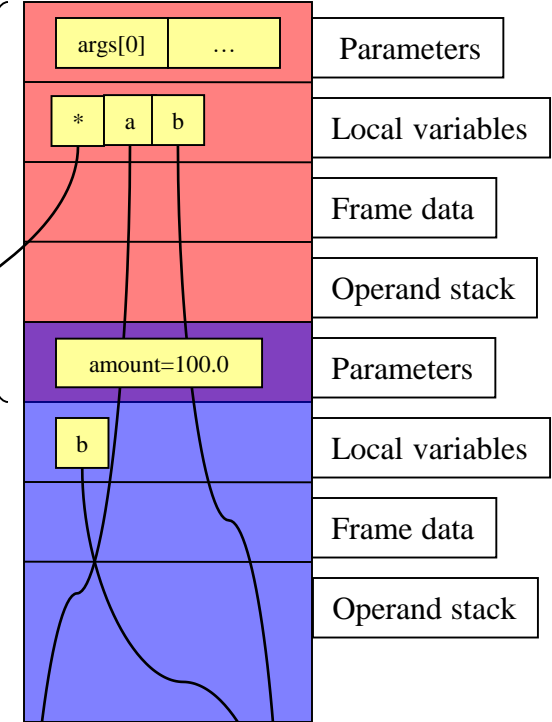
```
// BankAccount.java
private double balance;
private static int totalAccounts = 0;

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

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

**main()**

**BankAccount()**



# 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？

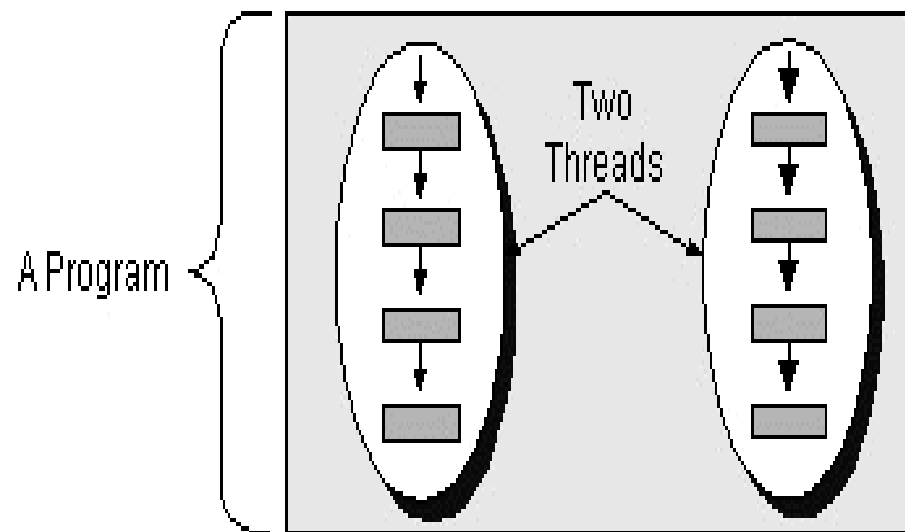
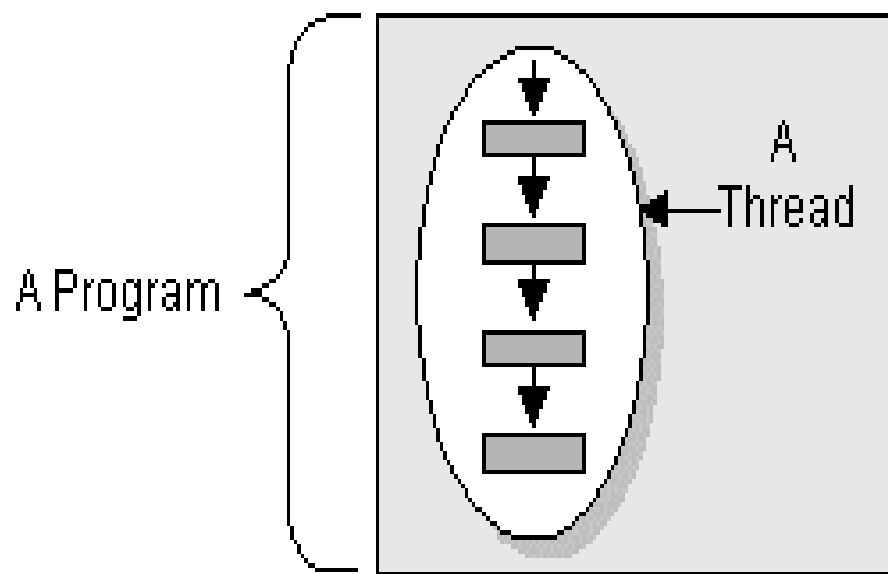
- 单线程的控制流程从main入口，直至main出口结束，中间经过一系列对象之间的相互协作→单一流程
- 如果单一流程中有些处理活动本身应该是并发的，这样的处理效率常常不能满足要求→多个并发流程→多个线程
  - CPU与OS都提供了并行/并发支持
- 任意两个并发流程cf1, cf2
  - 绝对并发：完全没有依赖关系
  - 部分并发：在特定条件下交换信息
- 识别并发流程的基本策略
  - 问题域中独立且并发活动的实体

# 多线程处理

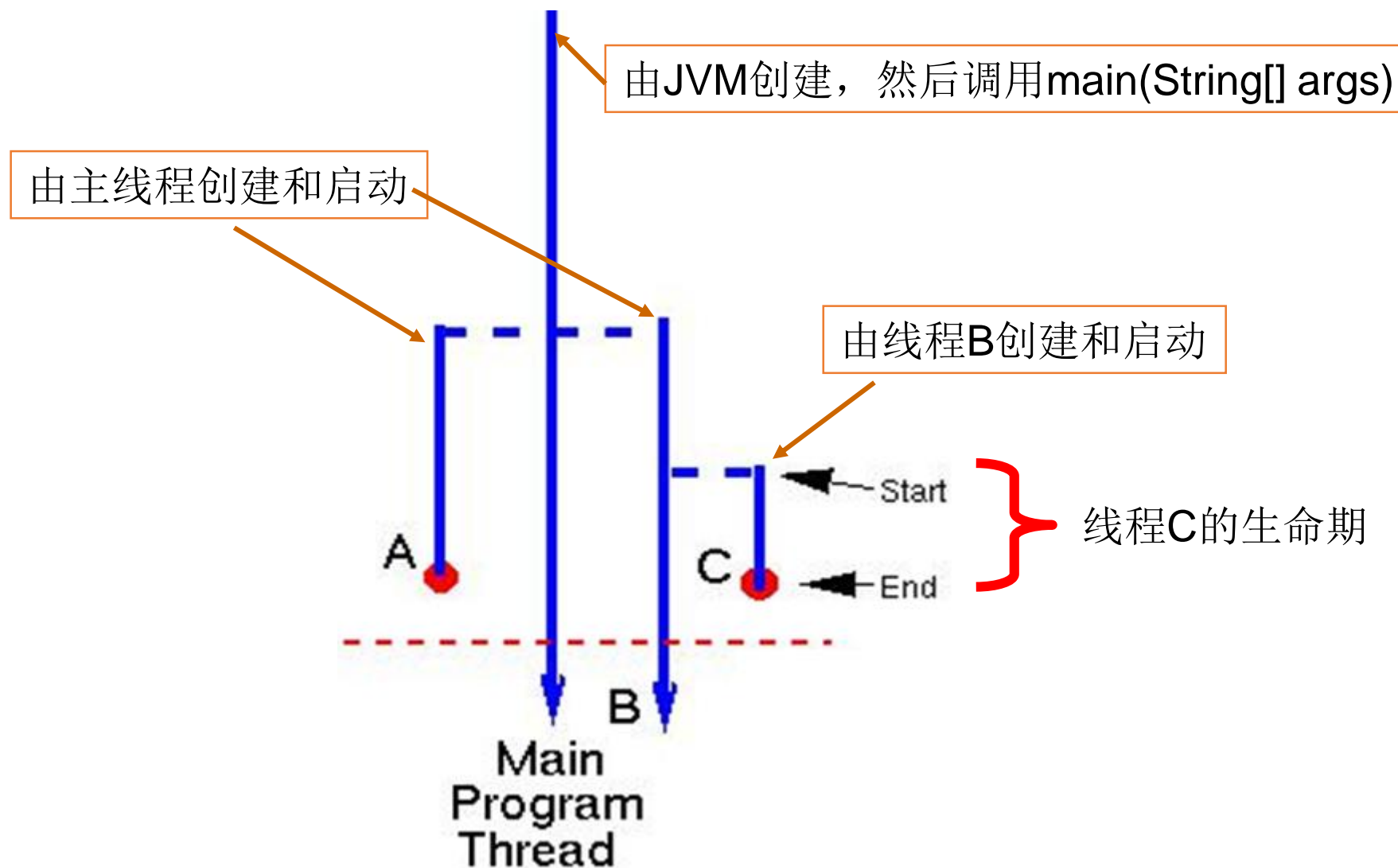
- 同步与异步
  - 同步：当一个方法执行返回时，调用者才能继续执行
  - 异步：不等方法执行返回，调用者继续执行
- 现实世界本质上是异步的
  - 每个对象同时在活动
  - 互相通知对方感兴趣的消息
  - 各自处理自己的消息
  - 在需要进行同步

# 多线程处理

- 线程：程序内的一个顺序执行控制(flow of control)单位(流)
- 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





# 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();

# 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)

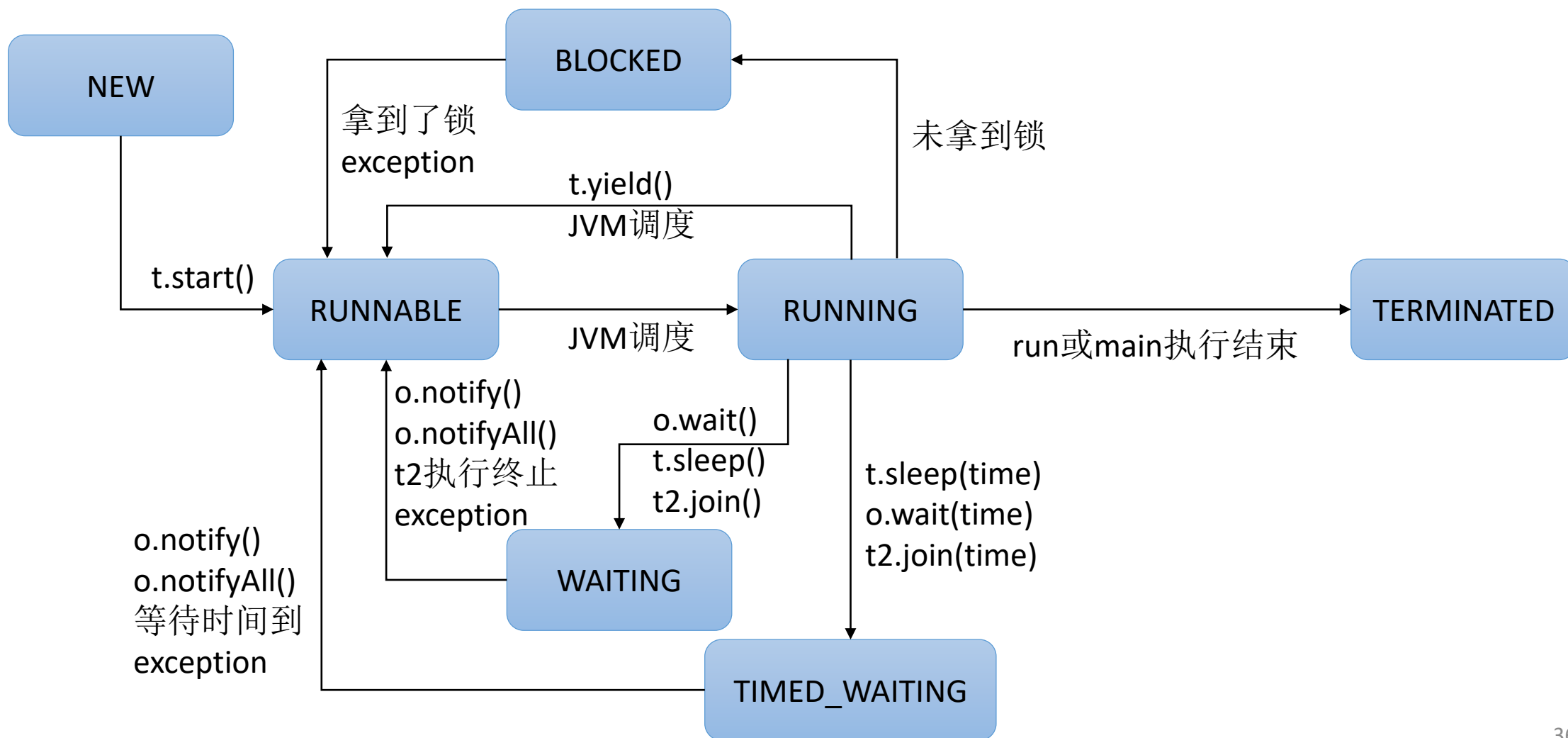
**BLOCKED**, // 阻塞状态, 无法获得公共数据访问或临界区执行权限

**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



# 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( ... ); // 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

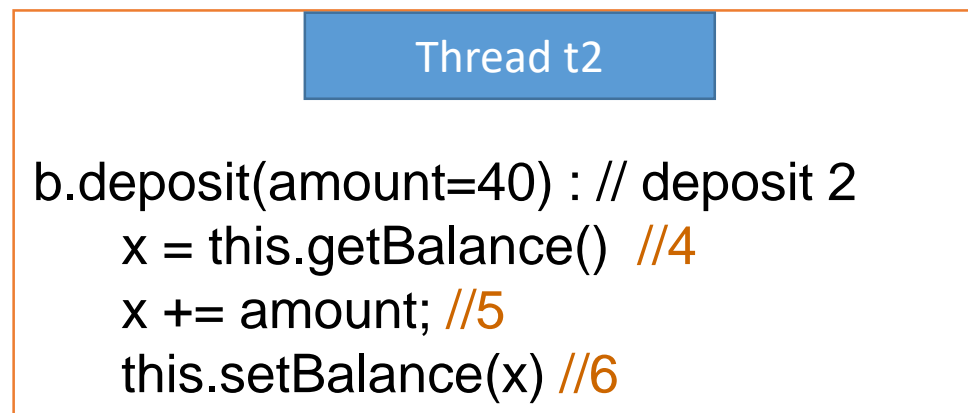
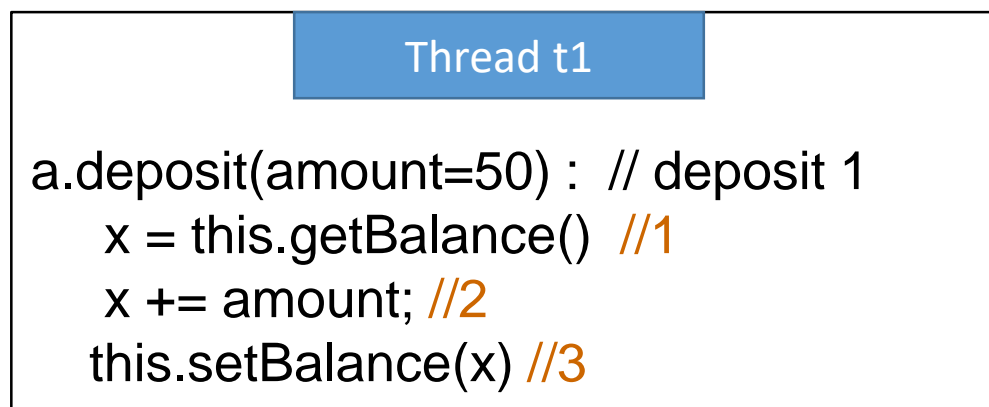
```
public class SimpleThread extends Thread {  
    public SimpleThread(String str) { super(str); }  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(i + " " + getName());  
            try {  
                sleep((long)(Math.random() * 1000));  
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



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？

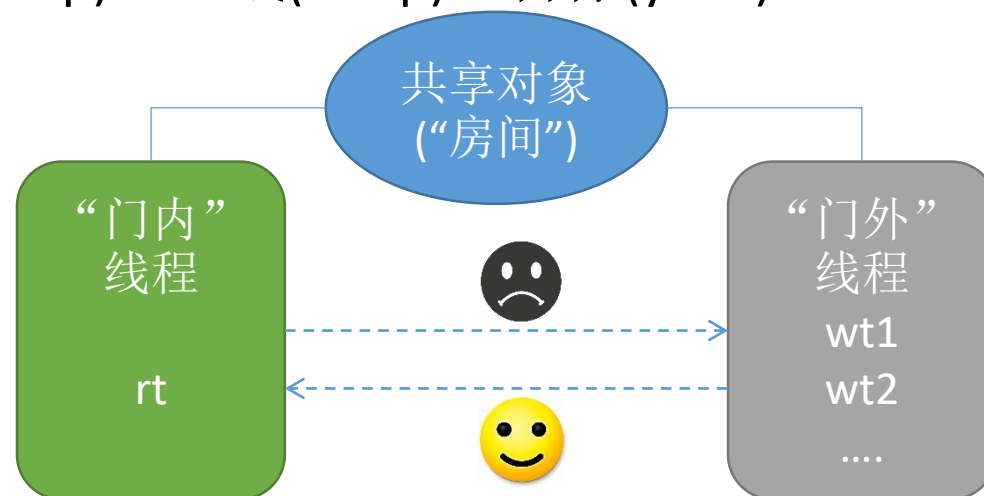
# 共享数据的访问控制

- 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



# 线程交互

- 多线程代码与单线程代码的重要差异
  - 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



# 线程的基本工作模式

- 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
  - 与“外面世界”进行数据交互的窗口
- 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
    - 对象锁: `synchronized(obj){}`
    - 类锁: `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

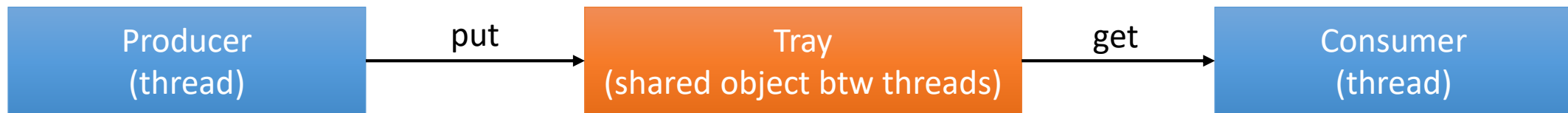
# 单线程程序与多线程程序的对比

单线程下线程处理流程	多线程下线程处理流程
单一流程	单一流程
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
不可以作为锁	可以作为锁
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

# Java多线程应用的典型样例

- 经典问题：生产者和消费者
  - 生产者向一个锁对象(托盘)里存入生产的货物 `//synchronized method`
  - 消费者从托盘里取走相应的货物 `//synchronized method`
  - 在货物被取走前，不能放入新的货物 `//控制变量表示托盘状态`
  - 在货物被取走后，不能再次取货 `//控制变量表示托盘状态`
  - 三个类：生产者、消费者、托盘



# Java多线程应用的典型样例

```
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() {  
        for (int i = 0; i < 10; i++)  
            for(int j =0; j < 10; j++ ) {  
                tray.put(i, j);  
                System.out.println("Producer #" + this.id  + " put: (" +i +", "+j + ").");  
                try { sleep((int)(Math.random() * 100)); }  
                catch (InterruptedException e) { }  
            };  
    }  
}
```

# 典型的Java多线程应用

```
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);  
        }  
    }  
}
```

# 典型的Java多线程应用

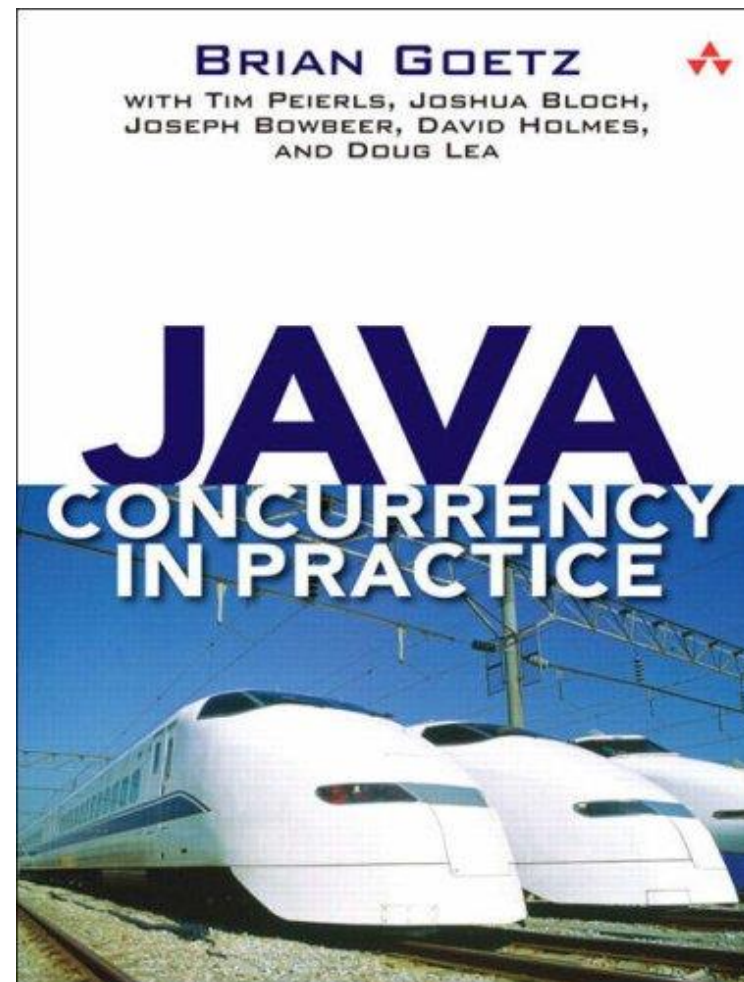
```
public class Tray {  
    private int x,y;    private boolean full = false;  
    public synchronized int get() {  
        while (full == false) {  
            try { wait(); } catch (InterruptedException e) { }  
            full = false;  
            // 此时full为true, 设为false后确保所有其他消费者不可能来抢  
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

synchronized  
notifyAll  
wait

# 典型的Java多线程应用

```
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();  
    }  
}
```

- 如何让生产者和消费者具有动态性？
  - 可以根据需要不断的生成内容
  - 只要生产者在生产，消费者就可以要消费
- 多线程程序有哪些潜在的问题？
- 如何确保一个类在多线程中的使用是安全的？
  - Thread-safe class

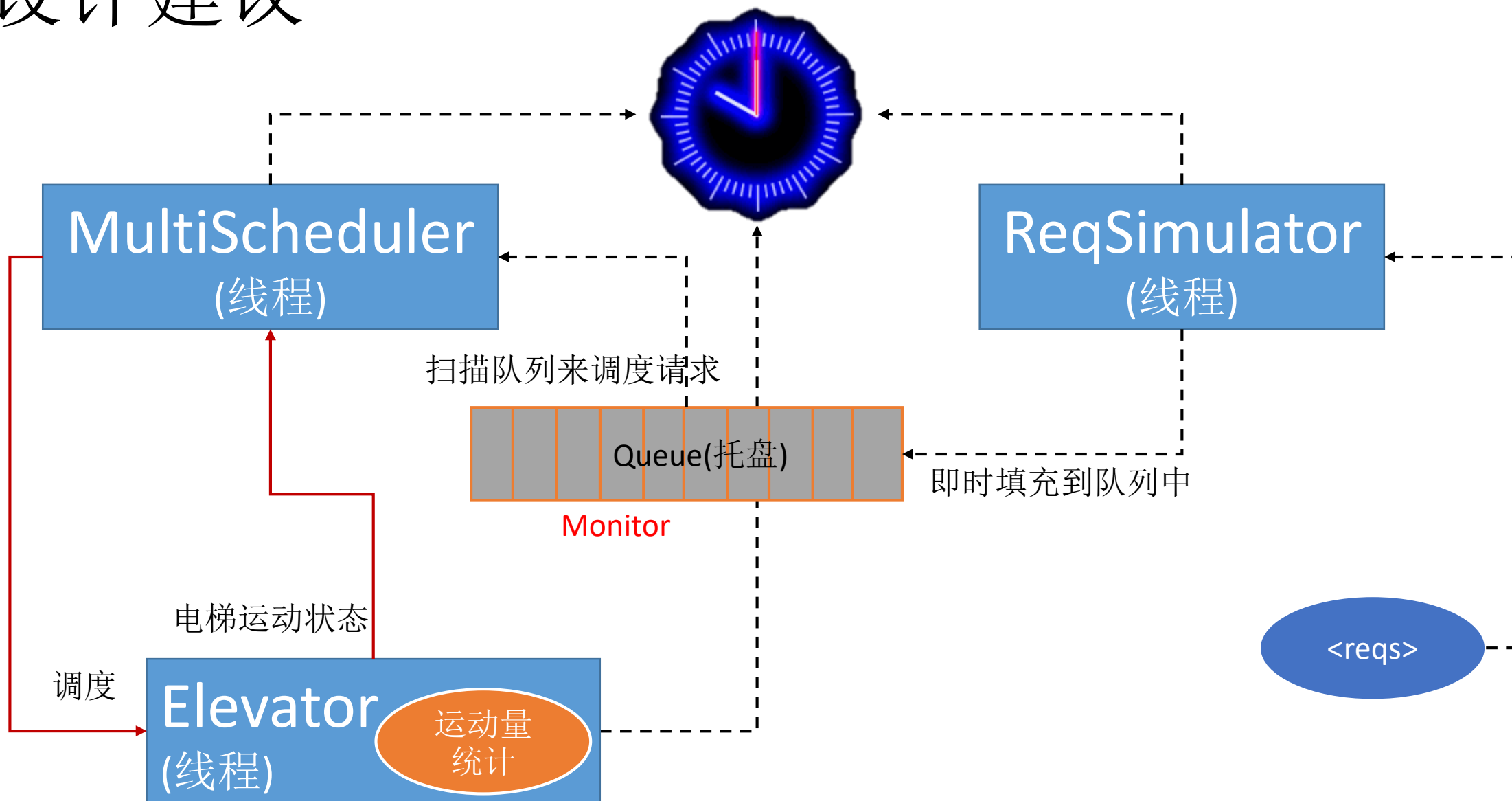




# 作业

- 使用多线程机制的多电梯系统(电梯数为3，楼层数为20)
  - 系统需要具有并发处理能力
    - 调度器综合调度多部电梯来消耗事件队列中的请求
    - 进一步模拟真实场景：请求产生时间自动从系统获取，系统运行时持续接受请求，并发对输入请求进行处理，填充事件队列
    - 请求的输入格式发生变化，时间 $t$ 自动从系统获得，按照100ms为单位来计算（不足100ms四舍五入）。系统启动时间点设为0。便于测试，电梯系统的时间控制从OS获得，电梯运行一层楼消耗3秒，开关门消耗6秒。
  - 支持多部电梯的调度
    - 统计电梯“运动量”：即电梯行驶的距离（楼层数）。
    - 运动量均衡的捎带调度策略：针对任何一个楼层请求，如果有多部电梯可以响应，优先选择进行捎带的电梯。如果有多部可以捎带，则选择运动量较小的电梯；如果没有可以捎带的电梯，则选择可以响应中的运动量较小的来响应。如果没有可以响应的电梯，则一直等待直至能够响应。
  - 程序输出
    - 同作业3，每部电梯独立输出
  - 设计要求
    - 参考后面的推荐设计和本次作业的要求来重构之前的设计。要求使用多线程，并继承作业3的Scheduler
    - 注意修复第三次作业的bug，否则会放大到让这次的程序crash。

# 设计建议



# 设计建议

