

Object Oriented Programming with Java

Zheng Chen



Multithreading

1. Java Thread
2. Thread life cycle
3. Synchronization
4. Pitfall and benefit



**KEEP
CALM
AND
CODE
JAVA**

活动监视器
所有进程

ⓧ

i

⋮

CPU内存能耗磁盘网络

🔍 搜索

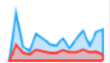
进程名称	% CPU	CPU 时间	% GPU	GPU 时间	PID	用户	端口	已发送字节	已接收字节	实际内存
WindowServer	17.9	28:50.76	0.0	10:02.25	132	_windowserver	3,312	0 字节	0 字节	92.6 MB
backupd	0.0	58:03.63	0.0	0.00	256	root	225	0 字节	0 字节	2.3 MB
📎 Keynote 讲演	0.0	28.85	0.0	0.00	12011	chenzheng	492	0 字节	0 字节	470.1 MB
kernel_task	17.1	47:49.59	0.0	0.00	0	root	0	0 字节	0 字节	1.84 GB
🌐 访达	0.2	13:06.19	0.0	0.00	363	chenzheng	1,154	0 字节	0 字节	143.3 MB
📠 微信	0.9	41.42	0.0	0.00	11728	chenzheng	1,021	32 KB	79 KB	170.0 MB
🔒 https://grammarly.com	0.0	3.14	0.0	0.00	11961	chenzheng	86	0 字节	0 字节	194.8 MB
🌐 Safari 浏览器	0.0	15.64	0.0	0.00	11944	chenzheng	625	0 字节	0 字节	148.0 MB
🖥️ 活动监视器	10.2	10.13	0.0	0.00	12098	chenzheng	2,250	0 字节	0 字节	125.0 MB
🔒 https://www.google.com	0.0	2.50	0.0	0.02	11960	chenzheng	101	0 字节	0 字节	151.3 MB
🔍 搜狗输入法	0.0	2:57.53	0.0	0.00	502	chenzheng	406	0 字节	0 字节	71.4 MB
com.apple.siri.embeddedspeech	0.0	17.01	0.0	0.00	442	chenzheng	68	0 字节	0 字节	6.9 MB
🖥️ 程序坞	0.1	41.16	0.0	0.00	358	chenzheng	796	0 字节	0 字节	28.9 MB
SogouTaskManager	0.0	2.95	0.0	0.00	11625	chenzheng	111	0 字节	0 字节	24.8 MB

系统:5.69%

用户:20.18%

闲置:74.14%

CPU 负载

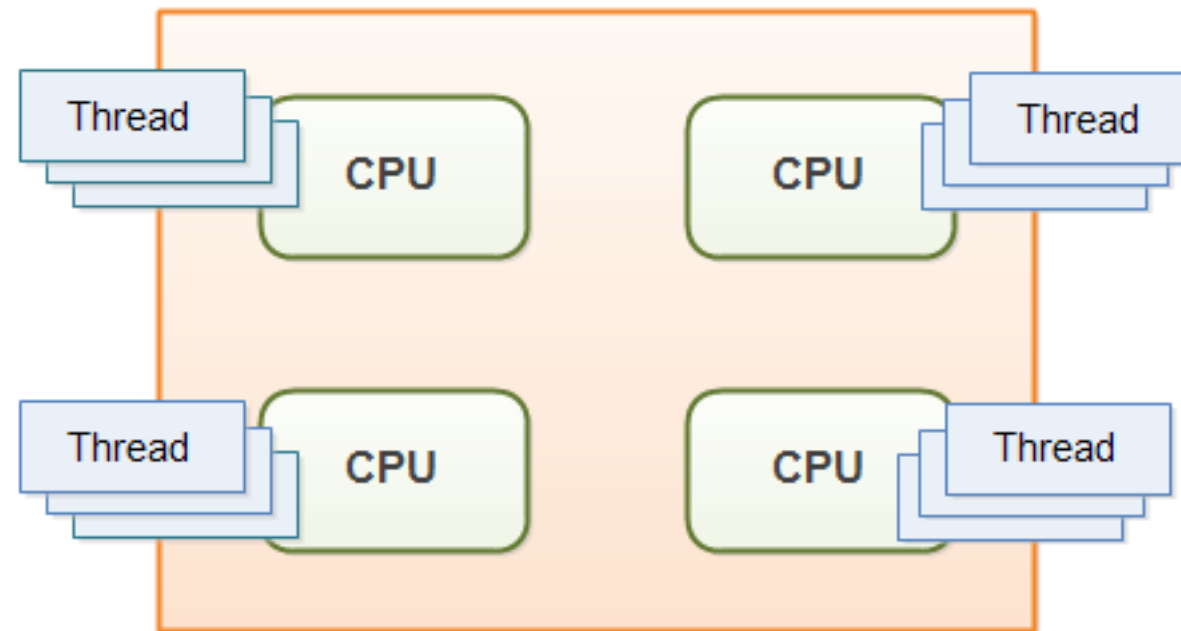


线程:1,646

进程:467

Thread

- A **thread** is a lightweight sub-**process**, the smallest unit of processing.



- Multiprocessing and multithreading, both are used to achieve multitasking. We use multithreading than multiprocessing because threads use a shared memory area.

Advantages of Multithreading

- You can perform many operations over different CPU cores, so it saves time.
- Threads are independent, so it doesn't affect other threads if an exception occurs in a single thread.
- It doesn't block the user because threads are independent and you can perform multiple operations at the same time.

Thread Class and Runnable Interface

- To create a new thread, your program will either extend the **Thread** class or implement the **Runnable** interface.

```
public class Thread extends Object implements Runnable{  
    //.....  
}
```

```
public interface Runnable{  
    void run()  
}
```

There are two ways to create a new thread of execution. One is to declare a class to be a subclass of Thread.

```
class PrimeThread extends Thread {
    long minPrime;
    PrimeThread(long minPrime) {
        this.minPrime = minPrime;
    }
    public void run() {
        // compute primes larger than minPrime
        . . .
    }
    public static void main(String[] args) {
        PrimeThread p = new PrimeThread(143);
        p.start();
    }
}
```

The other way to create a thread is to declare a class that implements the Runnable interface.

```
class PrimeRun implements Runnable {
    long minPrime;
    PrimeRun(long minPrime) {
        this.minPrime = minPrime;
    }
    public void run() {
        // compute primes larger than minPrime
        . . .
    }
    public static void main(String[] args) {
        PrimeRun p = new PrimeRun(143);
        new Thread(p).start();
    }
}
```


Multithreading

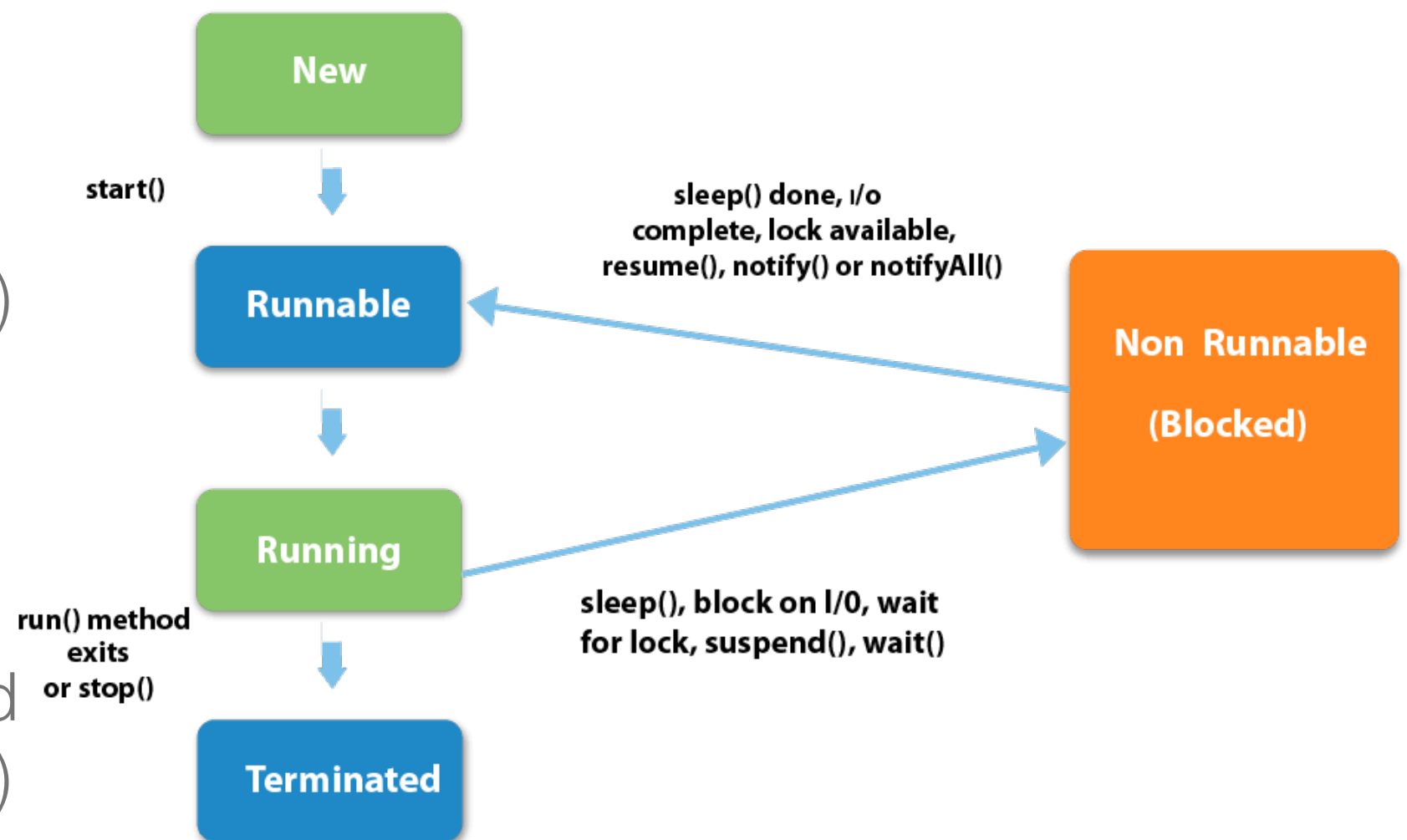
1. Java Thread
- 2. Thread life cycle**
3. Synchronization
4. Pitfall and benefit



**KEEP
CALM
AND
CODE
JAVA**

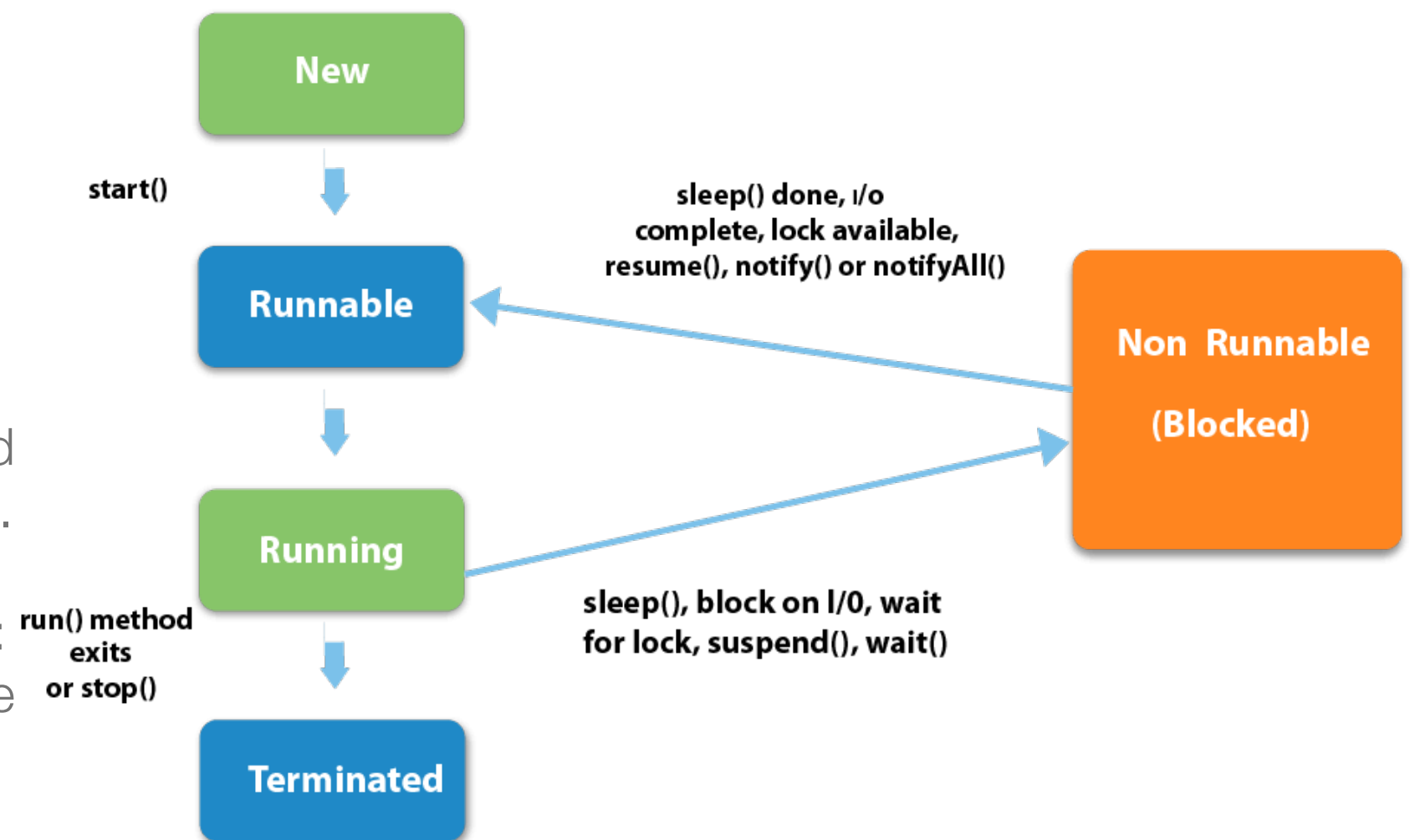
Thread Life Cycle in Java

- **New:** The thread is in new state if you create an instance of Thread class but before the invocation of start() method.
- **Terminated:** A thread is in terminated or dead state when its run() method exits.



Thread Life Cycle in Java

- **Runnable:** The thread is in runnable state after invocation of `start()` method, but the thread scheduler has not selected it to be the running thread.
- **Running:** The thread is in running state if the thread scheduler has selected it.
- **Non-Runnable (Blocked):** This is the state when the thread is still alive, but is currently not eligible to run.



Methods of Thread class

- public void **run**(): is used to perform action for a thread.
- public void **start**(): starts the execution of the thread. JVM calls the run() method on the thread.
- public void **sleep**(long milliseconds): Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- public void **join**(): waits for a thread to die.
- public void **join**(long milliseconds): waits for a thread to die for the specified milliseconds.
- public int **getPriority**(): returns the priority of the thread.
- public int **setPriority**(int priority): changes the priority of the thread.

Methods of Thread class

- public String **getName()**: returns the name of the thread.
- public void **setName**(String name): changes the name of the thread.
- public Thread **currentThread()**: returns the reference of currently executing thread.
- public int **getId()**: returns the id of the thread.
- public Thread.State **getState()**: returns the state of the thread.
- public boolean **isAlive()**: tests if the thread is alive.

Methods of Thread class

- public void **yield**(): causes the currently executing thread object to temporarily pause and allow other threads to execute.
- public void **suspend**(): is used to suspend the thread(deprecated).
- public void **resume**(): is used to resume the suspended thread(deprecated).
- public void **stop**(): is used to stop the thread(deprecated).
- public boolean **isDaemon**(): tests if the thread is a daemon thread.
- public void **setDaemon**(boolean b): marks the thread as daemon or user thread.
- public void **interrupt**(): interrupts the thread.
- public boolean **isInterrupted**(): tests if the thread has been interrupted.
- public static boolean **interrupted**(): tests if the current thread has been interrupted.

Thread Scheduler and Priority

- Thread scheduler in java is the part of the JVM that decides which thread should run.
- There is **no guarantee** that which runnable thread will be chosen to run by the thread scheduler.
- Each thread have a **priority**. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed.

3 constants defined in Thread class

```
public static int MIN_PRIORITY  
public static int NORM_PRIORITY  
public static int MAX_PRIORITY
```

- Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.
- A thread can call **setPriority**(int priority) to changes the priority of the thread.

Example of priority of a Thread

```
public class TestPriority extends Thread {

    public void run() {
        System.out.println("running thread:" + getName());
        System.out.println("priority is:" + getPriority());
    }

    public static void main(String args[]) {
        TestPriority m0 = new TestPriority();
        TestPriority m1 = new TestPriority();
        m0.setPriority(Thread.MIN_PRIORITY);
        m1.setPriority(Thread.MAX_PRIORITY);
        m0.start();
        m1.start();
    }
}
```

The sleep() method is used to sleep a thread for the specified amount of time.

```
public class TestSleep extends Thread {
    public void run() {
        for (int i = 0; i < 5; i++) {
            try {
                sleep(500);
            } catch (InterruptedException e) {
                System.out.println(e);
            }
            System.out.println(i + ": "
                               + System.currentTimeMillis());
        }
    }
    public static void main(String args[]) {
        new TestSleep().start();
    }
}
```

Call Thread.sleep() outside a Thread class

```
public class SleepOutside {  
    public static void main(String[] args)  
        throws InterruptedException {  
        long start = System.currentTimeMillis();  
        Thread.sleep(500);  
        System.out.println("Sleep time in ms = "  
            + (System.currentTimeMillis()  
                - start));  
    }  
}
```

join() method can be used to pause the current thread execution until the specified thread is dead.

```
public class TestJoin extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            try {
                Thread.sleep(500);
            } catch (Exception e) {
                System.out.println(e);
            }
            System.out.println(i + ": " + getName());
        }
    }
    public static void main(String args[]) throws InterruptedException {
        TestJoin t0 = new TestJoin();
        TestJoin t1 = new TestJoin();
        TestJoin t2 = new TestJoin();
        t0.start();
        t0.join();
        t1.start();
        t2.start();
    }
}
```

wait() and notify()

- wait — Object wait methods has three variance, one which waits indefinitely for any other thread to call notify or notifyAll method on the object to wake up the current thread. Other two variances puts the current thread in wait for specific amount of time before they wake up.
- notify — notify method wakes up only one thread waiting on the object and that thread starts execution. So if there are multiple threads waiting for an object, this method will wake up only one of them. The choice of the thread to wake depends on the OS implementation of thread management.
- notifyAll — notifyAll method wakes up all the threads waiting on the object, although which one will process first depends on the OS implementation.

```
class Message {  
    private String msg;  
    public Message(String str) {  
        this.msg = str;  
    }  
    public String getMsg() {  
        return msg;  
    }  
    public void setMsg(String str) {  
        this.msg = str;  
    }  
}
```

Wait notify example — Message

```

class Waiter implements Runnable {
    private Message msg;
    public Waiter(Message m) {
        this.msg = m;
    }
    public void run() {
        String name = Thread.currentThread().getName();
        synchronized (msg) {
            try {
                System.out.println(name + " waiting to get notified "
                    + "at time:" + System.currentTimeMillis());
                msg.wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println(name + " waiter thread got notified "
                + "at time:" + System.currentTimeMillis());
            //process the message now
            System.out.println(name + " processed: " + msg.getMsg());
        }
    }
}

```

Wait notify example — Waiter

```

class Notifier implements Runnable {
    private Message msg;
    public Notifier(Message msg) {
        this.msg = msg;
    }
    public void run() {
        String name = Thread.currentThread().getName();
        System.out.println(name + " started");
        try {
            Thread.sleep(1000);
            synchronized (msg) {
                msg.setMsg(name + " Notifier work done");
                // msg.notify(); only notify one thread.
                msg.notifyAll();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

Wait notify example — Notifier


```
public class WaitNotifyTest {  
    public static void main(String[] args) {  
        Message msg = new Message("process it");  
        Waiter waiter = new Waiter(msg);  
        new Thread(waiter, "waiter0").start();  
        Waiter waiter1 = new Waiter(msg);  
        new Thread(waiter1, "waiter1").start();  
        Notifier notifier = new Notifier(msg);  
        new Thread(notifier, "notifier").start();  
        System.out.println(  
            "All the threads are started");  
    }  
}
```

Wait notify example — WaitNotifyTest

Multithreading

1. Java Thread
2. Thread life cycle
3. **Synchronization**
4. Pitfall and benefit



KEEP
CALM
AND
CODE
JAVA

Synchronization

- Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.
- Synchronization in java is the capability to control the access of multiple threads to any shared resource.



The problem without Synchronization

```
class Account {
    int money;
    void withdraw(int amount) {
        money -= amount;
    }
    void deposit(int amount) {
        money += amount;
    }
}

class Customer extends Thread {
    Account account;
    Customer(Account account) {
        this.account = account;
    }
    public void run() {
        for (int i = 0; i < 10000; i++) {
            account.deposit(100);
            account.withdraw(100);
        }
    }
}
```

```
public class NoSynchronizationTest {
    public static void main(String[] args)
        throws InterruptedException {
        Account account = new Account();
        Thread t0 = new Customer(account);
        Thread t1 = new Customer(account);
        t0.start();
        t1.start();
        t0.join();
        t1.join();
        System.out.println(account.money);
    }
}
```

Synchronized method

- If you declare any method as synchronized, it is known as synchronized method.

```
synchronized methodName(){  
    ...  
}
```

- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Problem solved with Synchronized method

```
class Account {  
    int money;  
    synchronized void withdraw(int amount) {  
        money -= amount;  
    }  
    synchronized void deposit(int amount) {  
        money += amount;  
    }  
}
```

```
class Customer extends Thread {  
    Account account;  
    Customer(Account account) {  
        this.account = account;  
    }  
    public void run() {  
        for (int i = 0; i < 10000; i++) {  
            account.deposit(100);  
            account.withdraw(100);  
        }  
    }  
}
```

```
public class SynchronizationTest {  
    public static void main(String[] args)  
        throws InterruptedException {  
        Account account = new Account();  
        Thread t0 = new Customer(account);  
        Thread t1 = new Customer(account);  
        t0.start();  
        t1.start();  
        t0.join();  
        t1.join();  
        System.out.println(account.money);  
    }  
}
```

Synchronized block

- Synchronized blocks in Java are marked with the synchronized keyword.

```
synchronized(objectName){  
    ...  
}
```

- A synchronized block in Java is synchronized on some object.
- All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time.
- All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

Problem solved with Synchronized block

```
class Account {
    int money;
    void withdraw(int amount) {
        synchronized(this){
            money -= amount;
        }
    }
    void deposit(int amount) {
        synchronized(this){
            money += amount;
        }
    }
}

class Customer extends Thread {
    // the same....
}
```

```
public class AnotherSynchronizationTest {
    public static void main(String[] args)
        throws InterruptedException {
        Account account = new Account();
        Thread t0 = new Customer(account);
        Thread t1 = new Customer(account);
        t0.start();
        t1.start();
        t0.join();
        t1.join();
        System.out.println(account.money);
    }
}
```


Problem caused by Synchronized block

```
public class DeadLockTest {
    public static void main(String[] args) {
        DeadLockTest test = new DeadLockTest();
        final Object a = new Object();
        final Object b = new Object();
        Runnable block1 = new Runnable() {
            public void run() {
                synchronized (a) {
                    try {
                        // Adding delay
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                    // Thread-1 have a but need b also
                    synchronized (b) {
                        .....
                    }
                }
            }
        };
    }
}
```

```
        .....
        System.out.println("In block 1");
    }
}
};
Runnable block2 = new Runnable() {
    public void run() {
        synchronized (b) {
            // Thread-2 have b but need a also
            synchronized (a) {
                System.out.println("In block 2");
            }
        }
    }
};
new Thread(block1).start();
new Thread(block2).start();
}
}
```

Multithreading

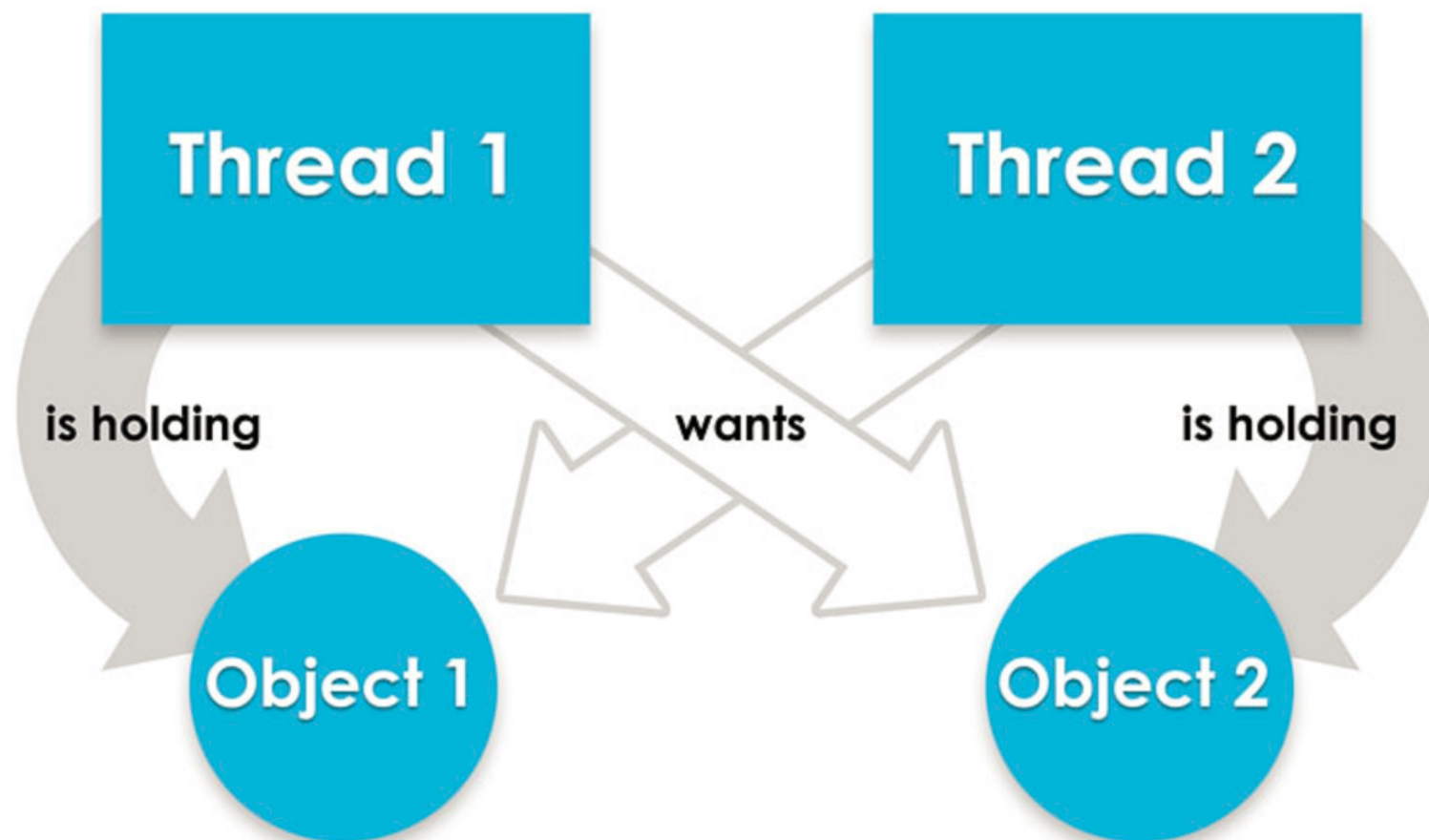
1. Java Thread
2. Thread life cycle
3. Synchronization
4. **Pitfall and benefit**



KEEP
CALM
AND
CODE
JAVA

Pitfall — Deadlock

- Deadlock is a situation where minimum two threads are holding the lock on some different resource, and both are waiting for other's resource to complete its task. And, none is able to leave the lock on the resource it is holding.



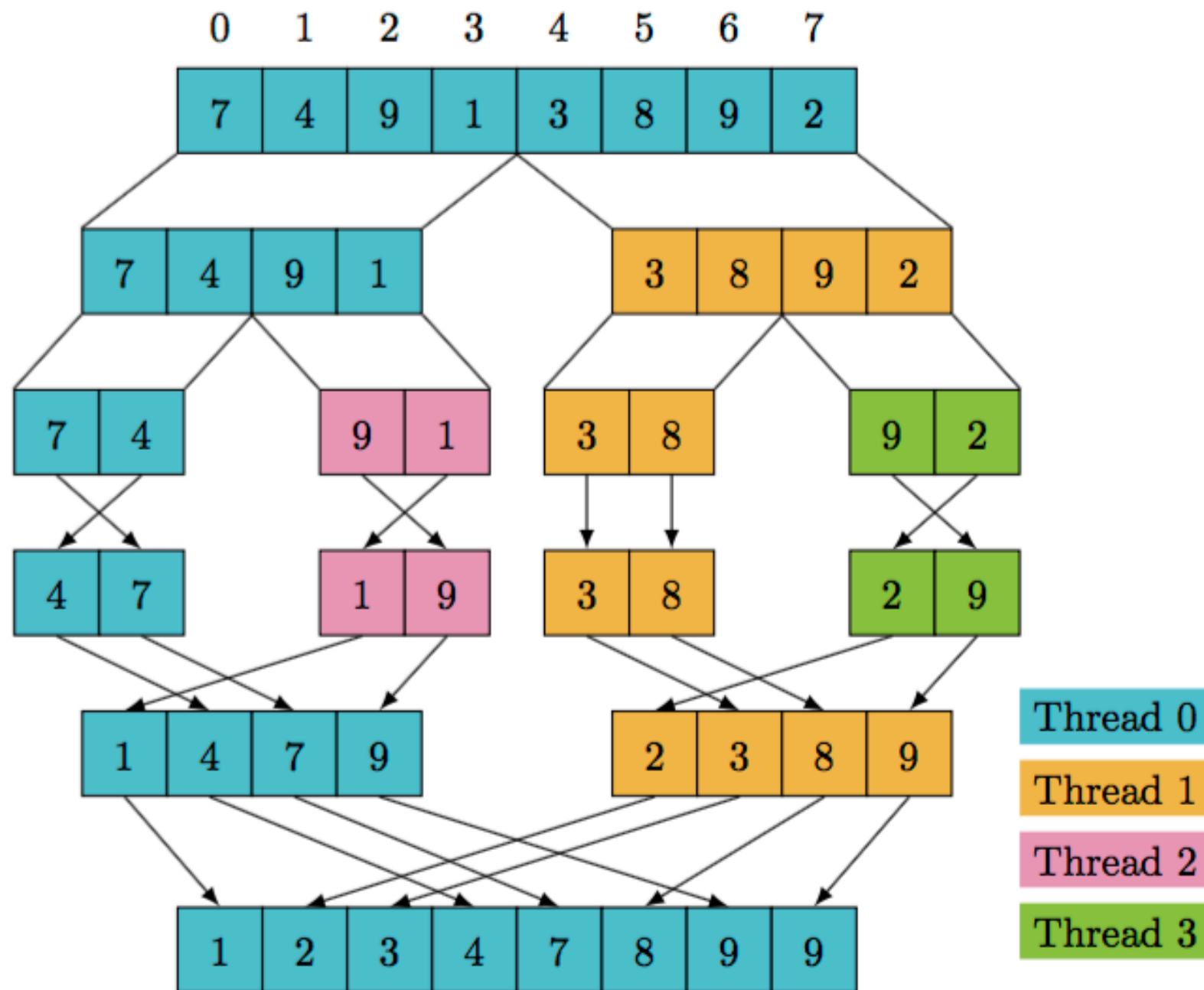
How to avoid deadlock?

- Deadlocks can be prevented by preventing at least one of the four required conditions:
 - Mutual Exclusion.
 - Hold and Wait.
 - No Preemption.
 - Circular Wait.

Benefit — Divide and Conquer

- Divide and Conquer is an algorithmic paradigm. Its basic idea is to decompose a given problem into two or more similar, but simpler, subproblems, to solve them in turn, and to compose their solutions to solve the given problem.
- A typical Divide and Conquer algorithm solves a problem using following three steps.
 - **Divide**: Break the given problem into subproblems of same type.
 - **Conquer**: Recursively solve these subproblems
 - **Combine**: Appropriately combine the answers

Multi-threaded merge sort



Homework

Implements a multi-threaded merge sort, and then compare the performance with the other sort program you did before.

