

# Computer System Design & Application

## 计算机系统设计与应用A

陶伊达 (TAO Yida)

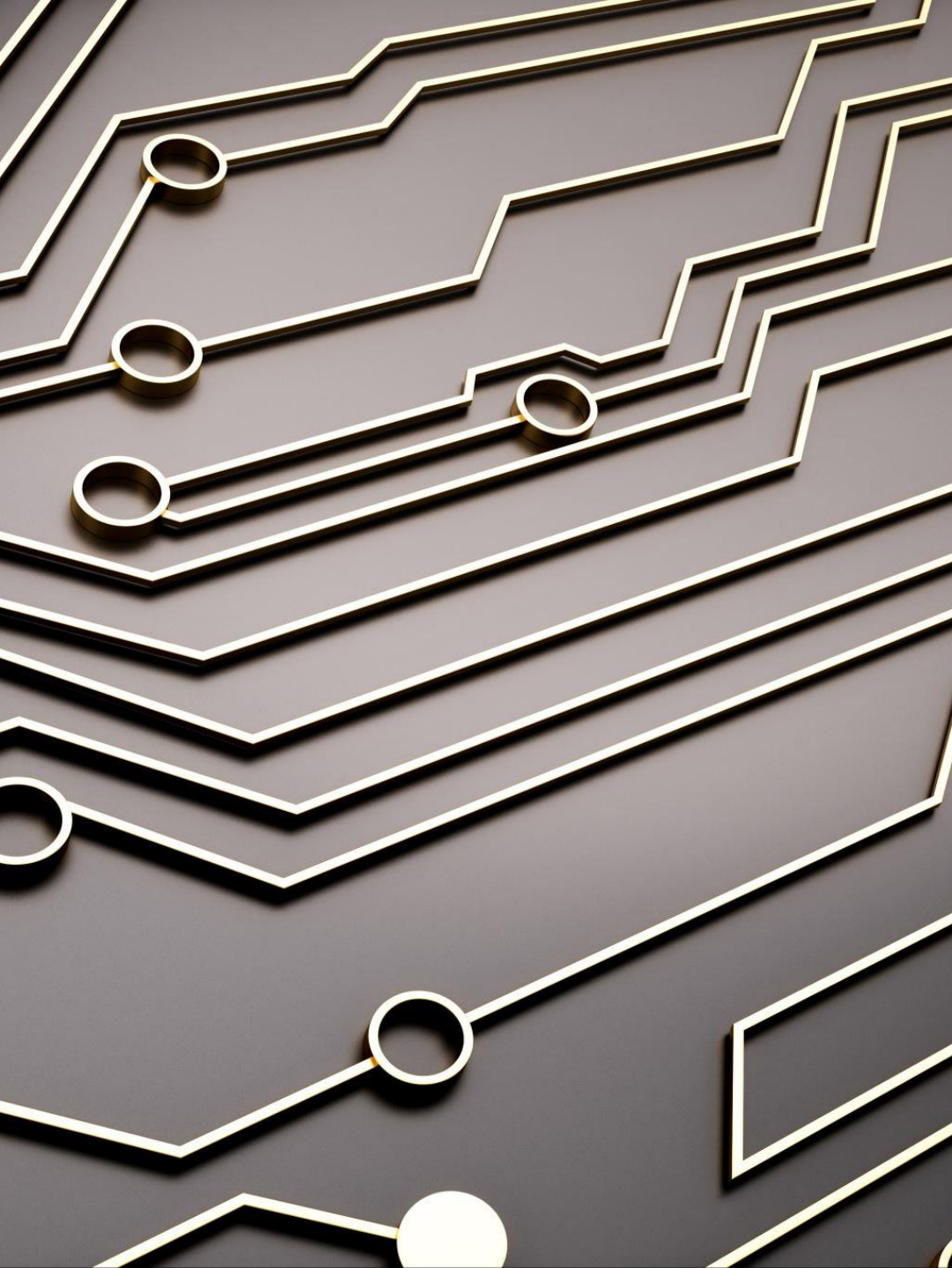
taoyd@sustech.edu.cn

An abstract graphic on the left side of the slide, featuring concentric circles and digital patterns in shades of blue, green, and white, resembling a stylized data visualization or a futuristic interface.

# Lecture 7

---

- Overview
- Creating & Starting Threads
- Synchronization
- Thread-safe Collections
- Tasks and Thread Pools

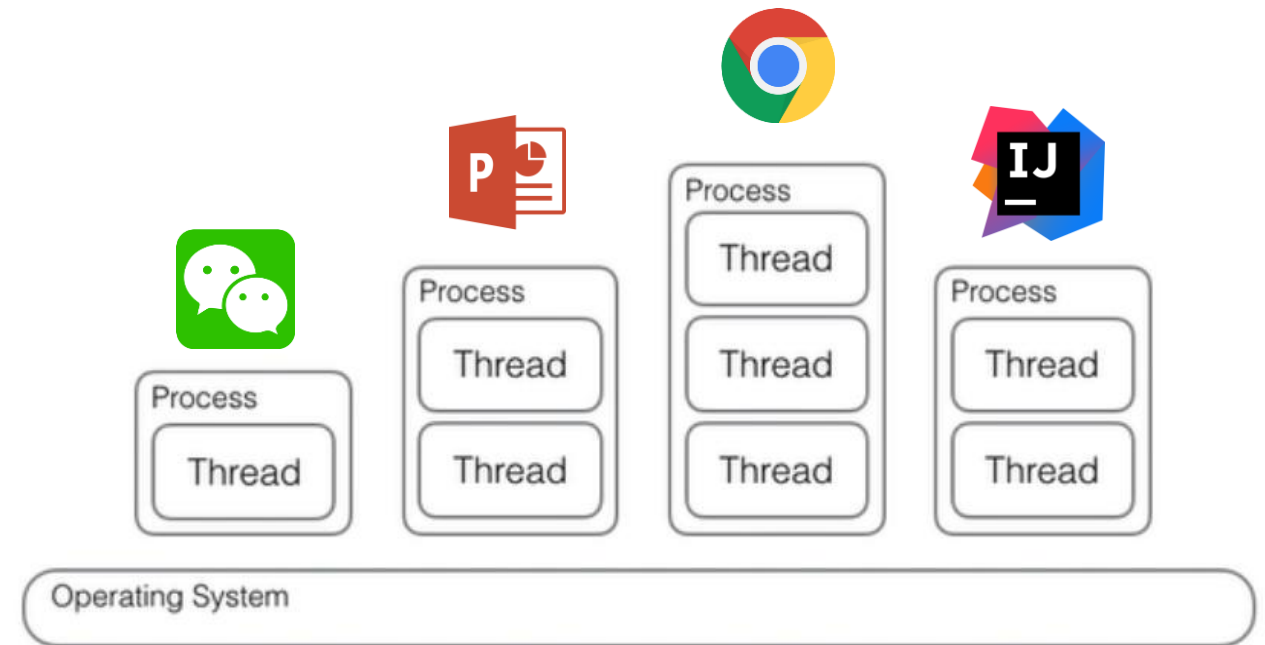


# Concurrent Programming

- Concurrency means multiple computations are happening at the same time
- In concurrent programming, there are two basic units of execution
  - Processes
  - Threads

# Process vs Thread

- **Process (进程)**
  - Executing a program starts a process (a running/active program)
  - OS allocates separate memory spaces for different processes
- **Thread (线程)**
  - A process can have multiple threads (at least 1 thread)
  - Threads within a process share the memory and resources of that process.



# Multithreading

5

- 
- In Java, concurrent programming is mostly concerned with threads
  - Multithreading refers to an individual program does **multiple tasks** at the same time
  - Each task is executed in a **thread**
  - Each thread defines a separate path of execution
    - The threads are independent, so it does not block the user to perform multiple operations at the same time
    - If an exception occurs in a single thread, it does not affect other threads.





# Lecture 7

---

- Overview
- Creating & Starting Threads
- Synchronization
- Thread-safe Collections
- Tasks and Thread Pools

# Multithreading in Java

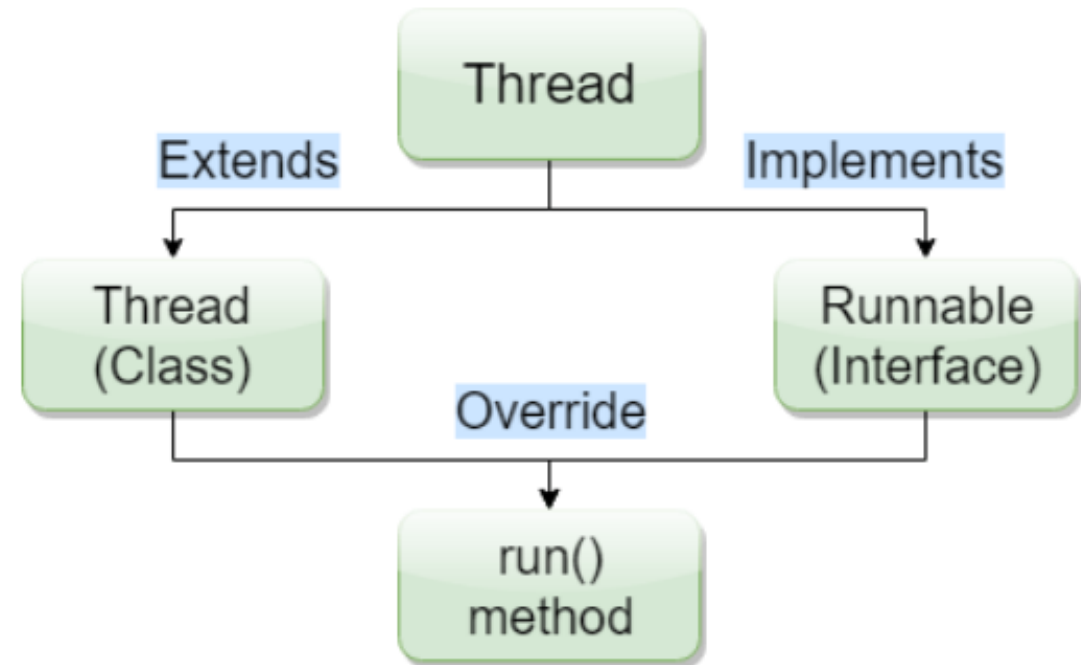
- JVM runs (mostly) as a single process
- The main thread is created automatically when our Java program is started
- The main thread has the ability to create additional threads

```
public class Concurrency {  
    public static void main(String[] args){  
        System.out.println(Thread.currentThread().getName());  
    }  
}
```

Output “main”

# Creating & Starting Threads

- Approach 1: Extending the Thread class **(not recommended)**
- Approach 2: Implementing the Runnable interface **(preferred)**





```
public class Cat extends Thread{
    @Override
    public void run(){
        System.out.printf("%s: cat\n", Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        Cat cat = new Cat();
        cat.start();
        System.out.printf("%s: main\n", Thread.currentThread().getName());
    }
}
```

## The Thread Class

- Declare a class to be a subclass of Thread
- This subclass should override the run method of Thread: specify what this thread does inside run().
- An instance of the subclass can then be allocated and started

# Using Thread

- Cat Thread
  - Print thread name and count for 10 times
  - Some interval between each print
- Main Thread
  - Print thread name and count for 10 times
  - Some interval between each print

```
public class CatThread extends Thread{
    int cnt = 0;

    @Override
    public void run(){
        while(cnt<10){
            System.out.printf("Thread %s: Cat %d\n",
                               Thread.currentThread().getName(), ++cnt);
            try{
                Thread.sleep(100);
            }catch (InterruptedException e){
                e.printStackTrace();
            }
        }
    }

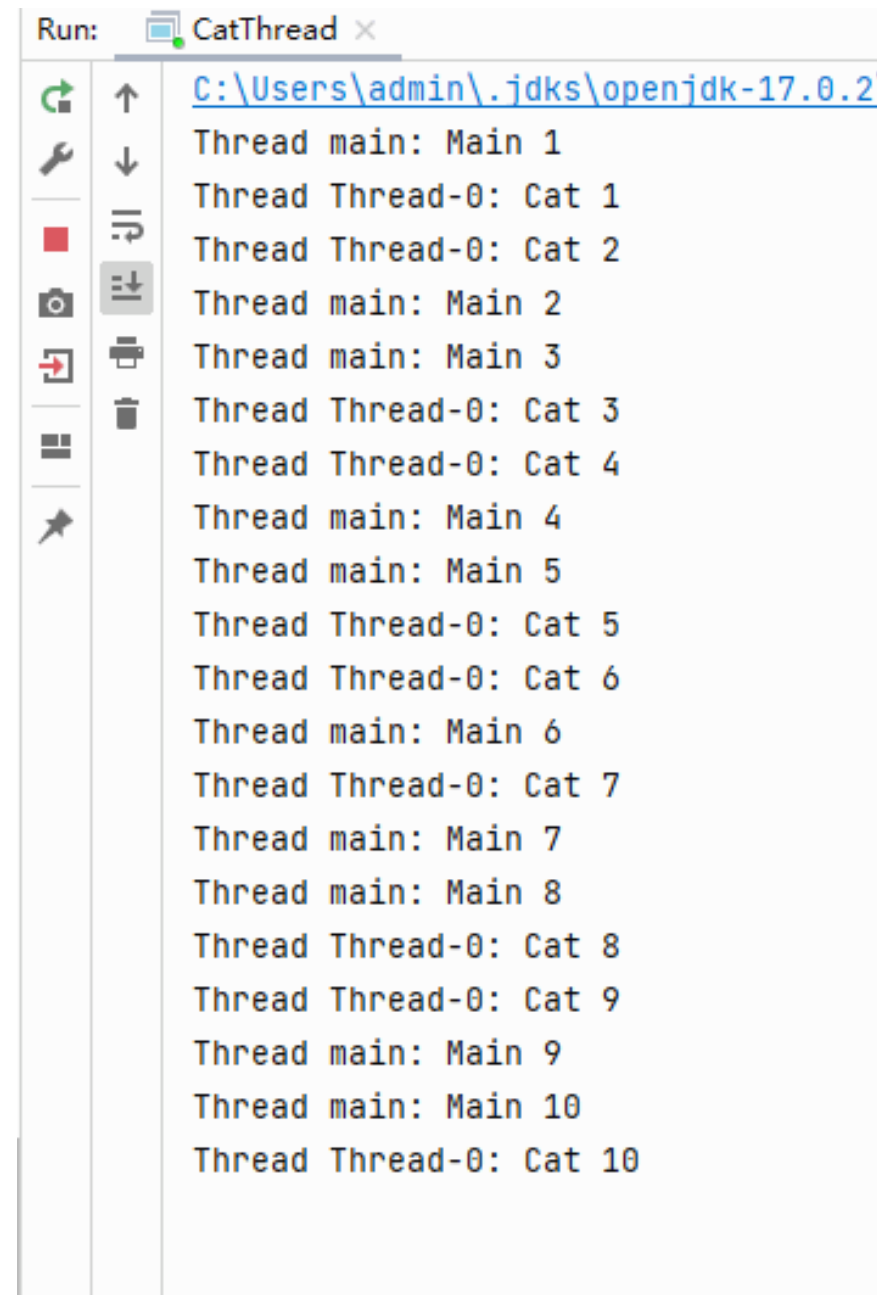
    public static void main(String[] args) throws InterruptedException {

        Thread cat = new CatThread();
        cat.start();

        int cnt=0;
        while(cnt<10){
            System.out.printf("Thread %s: Main %d\n",
                               Thread.currentThread().getName(), ++cnt);
            Thread.sleep(100);
        }
    }
}
```

# Using Thread

- Cat Thread
  - Print thread name and count for 10 times
  - 1s interval between each print
- Main Thread
  - Print thread name and count for 10 times
  - 1s interval between each print

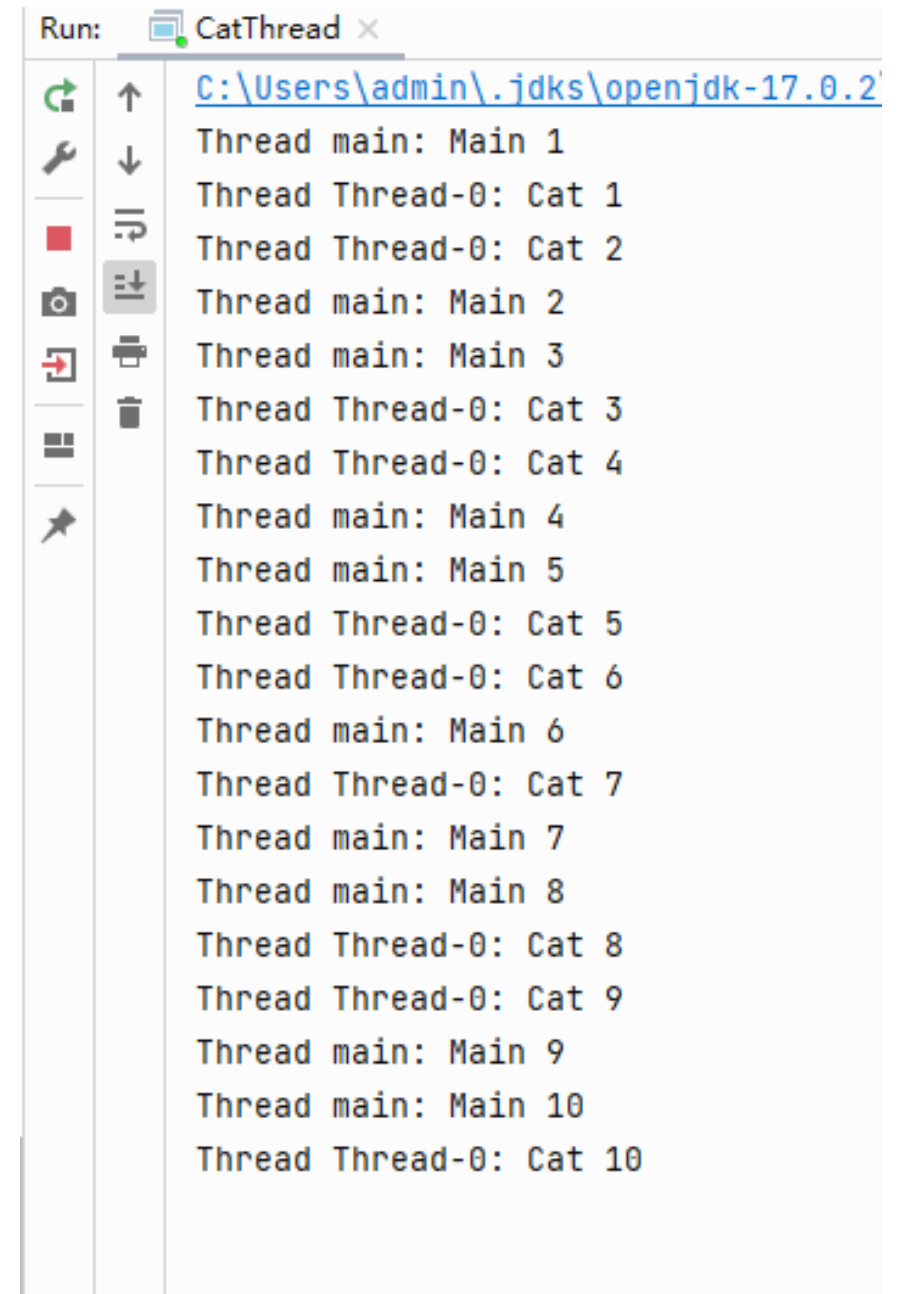


```
Run: CatThread x
C:\Users\admin\.jdk\openjdk-17.0.2
Thread main: Main 1
Thread Thread-0: Cat 1
Thread Thread-0: Cat 2
Thread main: Main 2
Thread main: Main 3
Thread Thread-0: Cat 3
Thread Thread-0: Cat 4
Thread main: Main 4
Thread main: Main 5
Thread Thread-0: Cat 5
Thread Thread-0: Cat 6
Thread main: Main 6
Thread Thread-0: Cat 7
Thread main: Main 7
Thread main: Main 8
Thread Thread-0: Cat 8
Thread Thread-0: Cat 9
Thread main: Main 9
Thread main: Main 10
Thread Thread-0: Cat 10
```

# Using Thread

The print operations for the Cat thread and the main thread are executed simultaneously

- Try execute the same program multiple times. Do we always get the same results?
- Try change the sleep duration. What will happen?

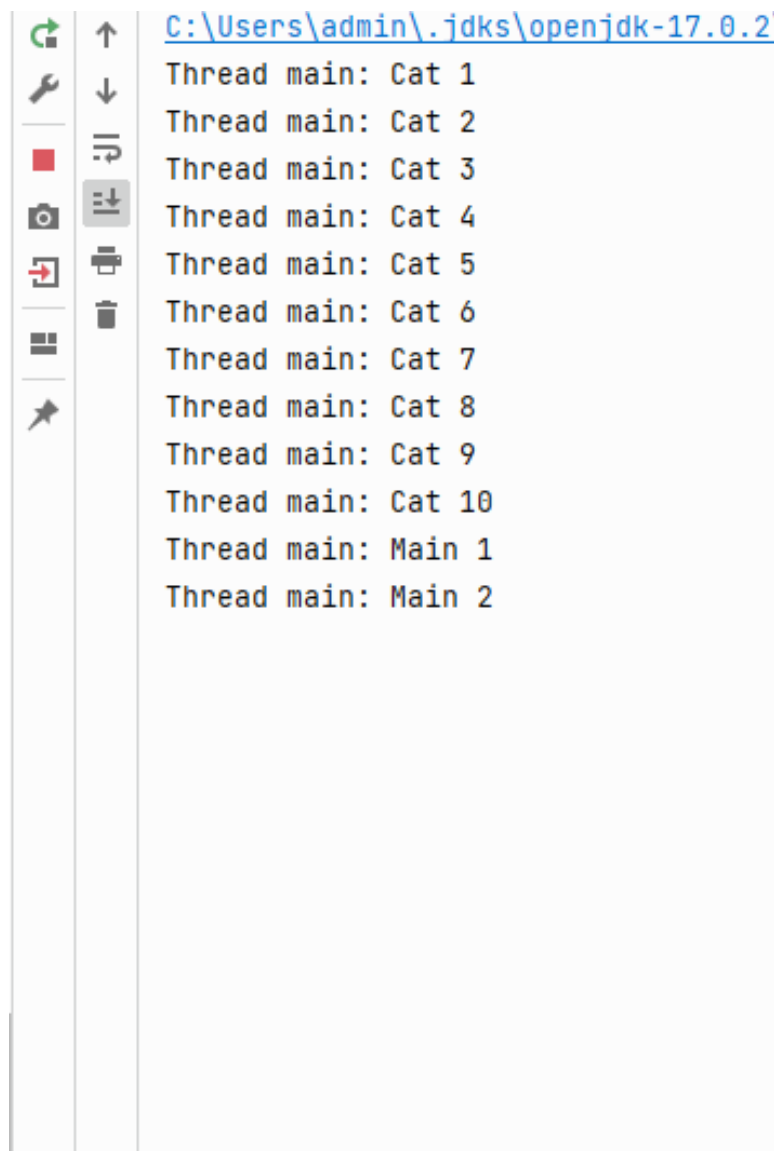


```
Run: CatThread x
C:\Users\admin\.jdk\openjdk-17.0.2
Thread main: Main 1
Thread Thread-0: Cat 1
Thread Thread-0: Cat 2
Thread main: Main 2
Thread main: Main 3
Thread Thread-0: Cat 3
Thread Thread-0: Cat 4
Thread main: Main 4
Thread main: Main 5
Thread Thread-0: Cat 5
Thread Thread-0: Cat 6
Thread main: Main 6
Thread Thread-0: Cat 7
Thread main: Main 7
Thread main: Main 8
Thread Thread-0: Cat 8
Thread Thread-0: Cat 9
Thread main: Main 9
Thread main: Main 10
Thread Thread-0: Cat 10
```

# Why start() instead of run()?

```
Thread cat = new CatThread();  
//cat.start();  
cat.run();
```

1. Things are executed sequentially instead of simultaneously
2. There is even **no** Cat thread!

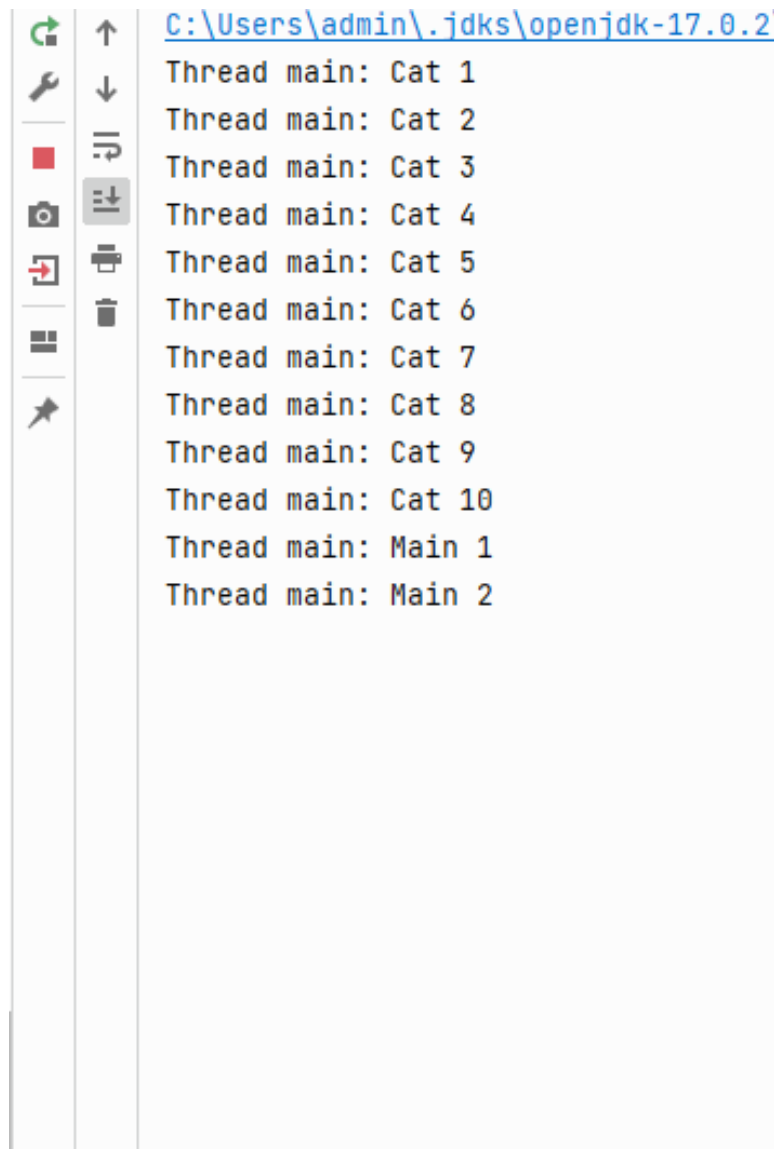
A screenshot of a Java IDE window. The title bar shows the file path 'C:\Users\admin\jdk\openjdk-17.0.2'. The main text area displays the output of a program, listing threads in a sequential order: 'Thread main: Cat 1' through 'Thread main: Cat 10', followed by 'Thread main: Main 1' and 'Thread main: Main 2'. On the left side of the IDE, there is a vertical toolbar with various icons for running, debugging, and other IDE functions.

```
C:\Users\admin\jdk\openjdk-17.0.2  
Thread main: Cat 1  
Thread main: Cat 2  
Thread main: Cat 3  
Thread main: Cat 4  
Thread main: Cat 5  
Thread main: Cat 6  
Thread main: Cat 7  
Thread main: Cat 8  
Thread main: Cat 9  
Thread main: Cat 10  
Thread main: Main 1  
Thread main: Main 2
```



# Why start() instead of run()?

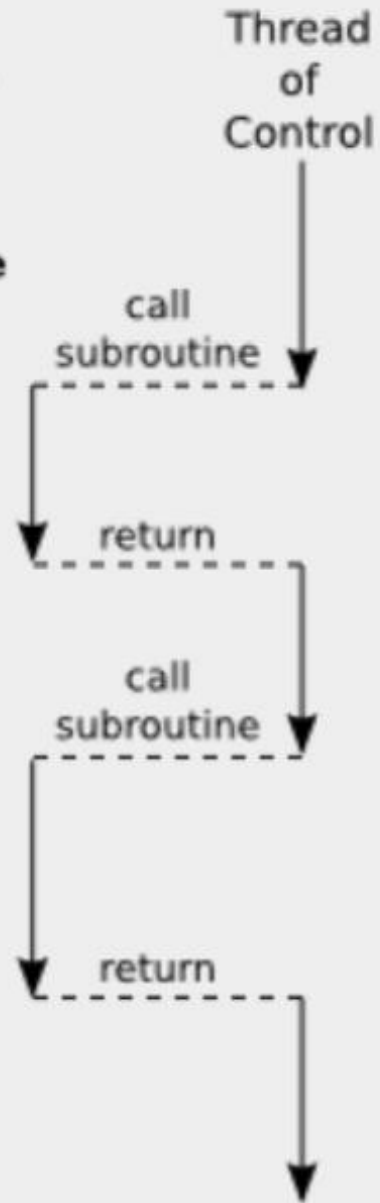
- `run()` executes like a normal method in the current thread
- `start()` indeed creates a new thread then calls `run()`
- `start()` is non-blocking: don't have to wait for it before executing the subsequent operations

A screenshot of a Java IDE window. The title bar shows the file path 'C:\Users\admin\jdk\openjdk-17.0.2'. The main text area contains the following output:

```
Thread main: Cat 1
Thread main: Cat 2
Thread main: Cat 3
Thread main: Cat 4
Thread main: Cat 5
Thread main: Cat 6
Thread main: Cat 7
Thread main: Cat 8
Thread main: Cat 9
Thread main: Cat 10
Thread main: Main 1
Thread main: Main 2
```

The IDE interface includes a toolbar on the left with icons for running, debugging, and other development tools.

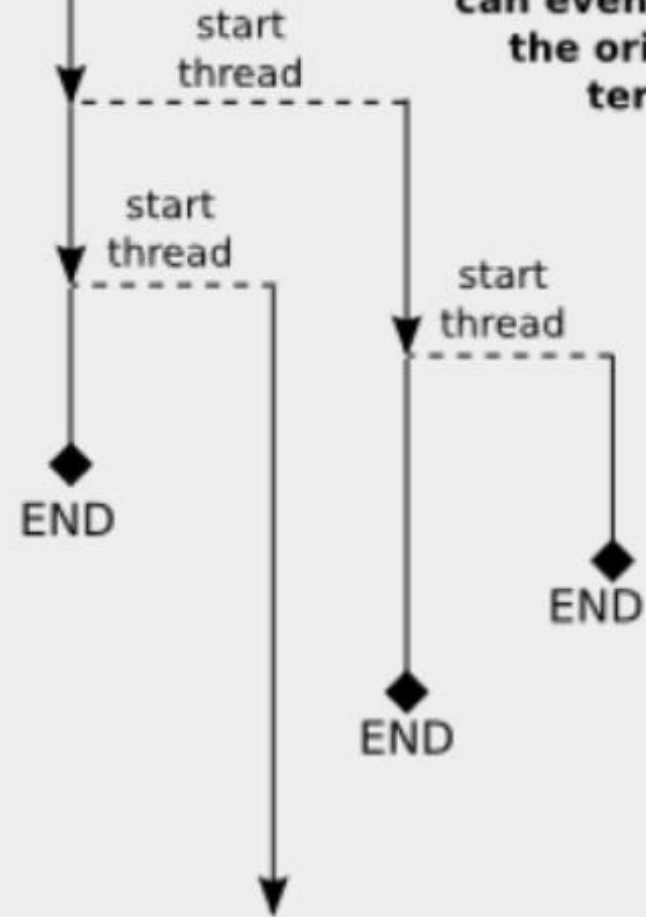
When a thread calls a subroutine, there is still only one thread of control, which is in the subroutine for a time until the subroutine returns.



Time



Thread of Control



When a thread starts another thread, there is a new thread of control that runs in parallel with the original thread of control, and can even continue after the original thread terminates.

## Flow of Control

```
@FunctionalInterface
```

```
public interface Runnable {
```

When an object implementing interface `Runnable` is used to create a thread, starting the thread causes the object's `run` method to be called in that separately executing thread.

The general contract of the method `run` is that it may take any action whatsoever.

See Also: `Thread.run()`

```
public abstract void run();
```

```
}
```

## The Runnable Interface

- The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread ( `Thread` class also does so)
- To implement `Runnable`, a class must implement the abstract method `run()`

# Starting a Thread with a Runnable

- Thread has a constructor that takes a Runnable

`Thread(Runnable target)`  
Allocates a new Thread object.

- To have the run() method executed by a thread, pass an instance of a class, anonymous class or lambda expression that implements the Runnable interface to a Thread constructor

```
Runnable cat = () -> System.out.println("Cat thread");
```

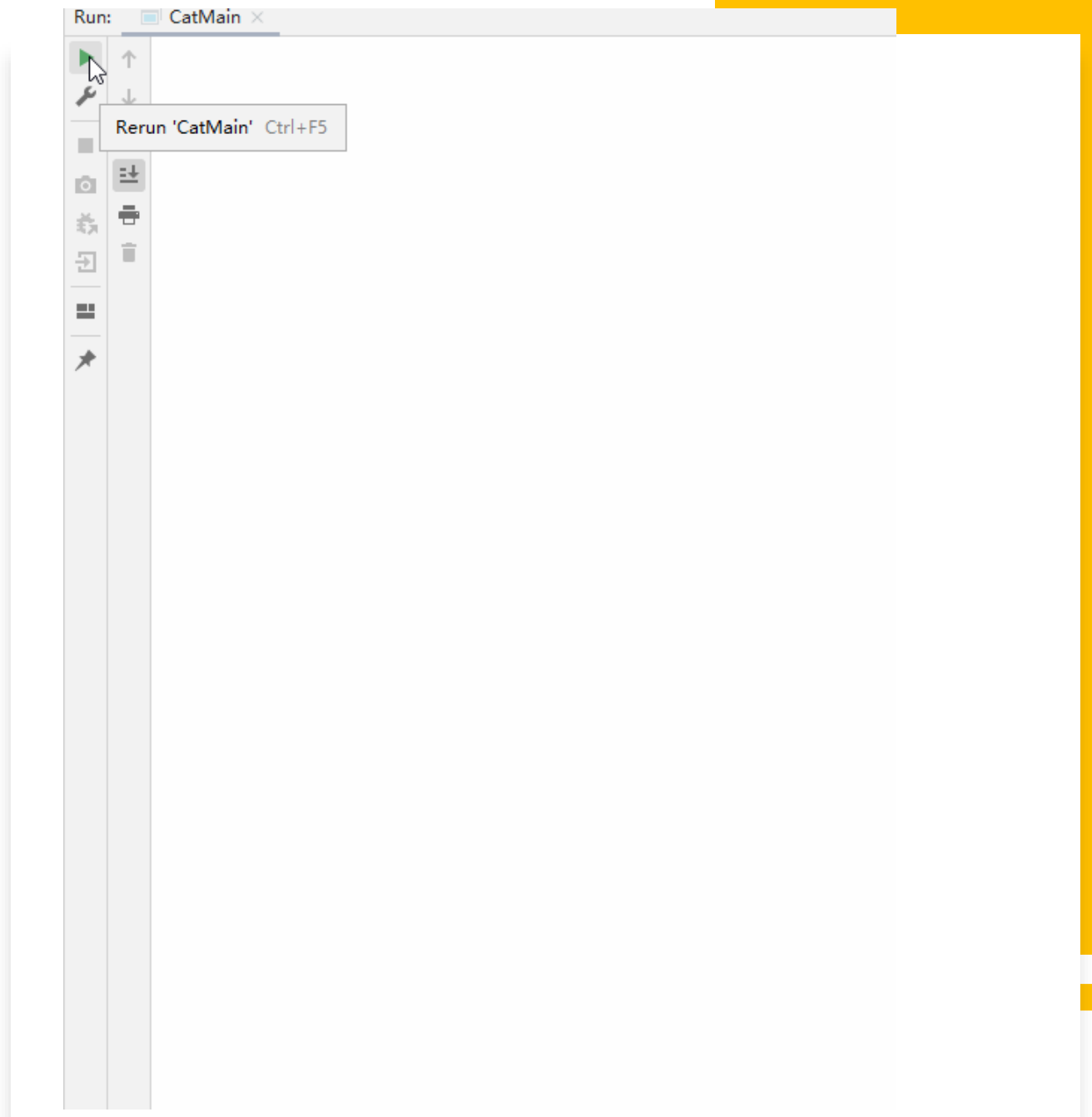
```
Thread thread = new Thread(cat);  
thread.start();
```

# Subclass vs Runnable

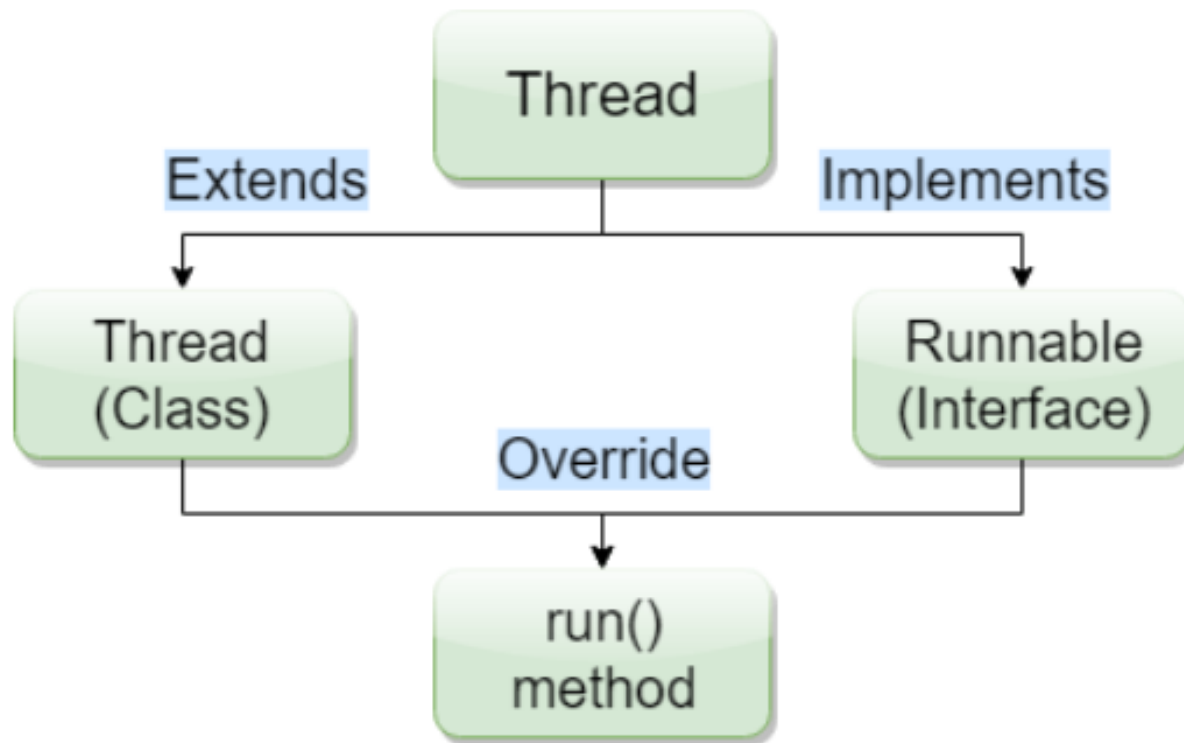
```
public static void main(String[] args) throws InterruptedException {  
    // Cat thread (subclassing)  
    Thread cat = new CatThread();  
    cat.start();  
  
    // Dog thread (runnable)  
    Runnable runnable = () -> {  
        int cnt = 0;  
        while(cnt < 10) {  
            System.out.printf("Thread %s: Dog %d\n",  
                               Thread.currentThread().getName(), ++cnt);  
            try {  
                Thread.sleep(500);  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    };  
    Thread dog = new Thread(runnable);  
    dog.start();  
  
    // Main thread  
    int cnt=0;  
    while(cnt < 10){  
        System.out.printf("Thread %s: Main %d\n",  
                           Thread.currentThread().getName(), ++cnt);  
        Thread.sleep(500);  
    }  
}
```



# Subclass vs Runnable



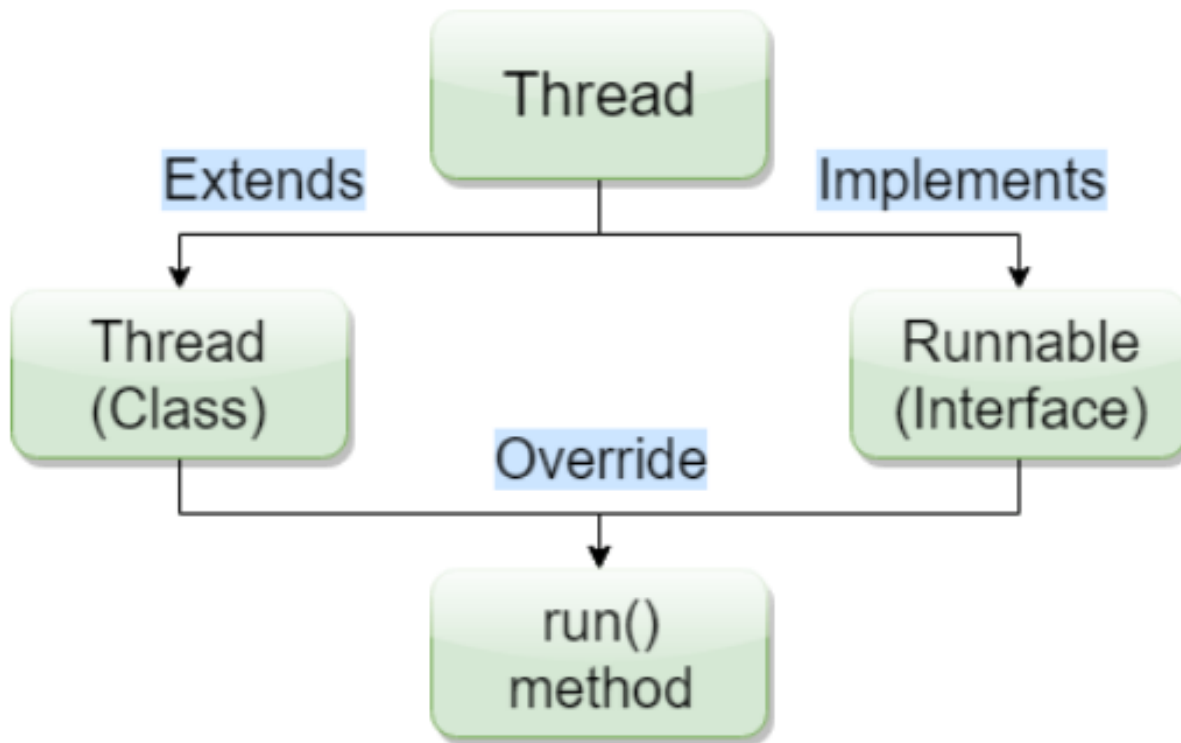
# Subclass vs Runnable



## Practical POV

- Java doesn't support multiple inheritances.
- If a class extends `Thread`, it cannot extend other classes
- If a class implements `Runnable`, it can still extend other classes

# Subclass vs Runnable



## Design POV

- In OOP, extending a class generally means adding new functionality and modifying/improving behaviors
- But we're not really improving a thread's behavior, we're just giving it something to run (task)
- Implementing Runnable **decouple** the task from its execution



# Lecture 7

---

- Overview
- Creating & Starting Threads
- Synchronization
- Thread-safe Collections
- Tasks and Thread Pools

# Problem: Bank Transfer

```
public class BankAccount {
    private double balance;

    public BankAccount(){
        balance = 0;
    }

    public void withdraw(double amount){
        balance -= amount;
        System.out.printf("Withdraw: %.1f, " +
            "new balance: %.1f\n", amount, balance);
    }

    public void deposit(double amount){
        balance += amount;
        System.out.printf("Deposit: %.1f, " +
            "new balance: %.1f\n", amount, balance);
    }
}
```

```
public class BankTransfer {
    public static void main(String[] args) {
        BankAccount account = new BankAccount();

        Runnable depositActivity = () -> {
            account.deposit( amount: 100);
        };

        Runnable withdrawActivity = () -> {
            account.withdraw( amount: 100);
        };

        for(int threads = 1; threads <= 10; threads++){
            Thread depositThread = new Thread(depositActivity);
            Thread withdrawThread = new Thread(withdrawActivity);

            depositThread.start();
            withdrawThread.start();
        }
    }
}
```



# Problem: Bank Transfer

```
public class BankAccount {
    private double balance;

    public BankAccount(){
        balance = 0;
    }

    public void withdraw(double amount){
        balance -= amount;
        System.out.printf("Withdraw: %.1f, " +
            "new balance: %.1f\n", amount, balance);
    }

    public void deposit(double amount){
        balance += amount;
        System.out.printf("Deposit: %.1f, " +
            "new balance: %.1f\n", amount, balance);
    }
}
```

```
Deposit: 100.0, new balance: 100.0
Deposit: 100.0, new balance: 0.0
Deposit: 100.0, new balance: -100.0
Withdraw: 100.0, new balance: -200.0
Withdraw: 100.0, new balance: -100.0
Deposit: 100.0, new balance: 0.0
Withdraw: 100.0, new balance: -100.0
Deposit: 100.0, new balance: 0.0
Withdraw: 100.0, new balance: -100.0
Deposit: 100.0, new balance: 0.0
Withdraw: 100.0, new balance: -100.0
Deposit: 100.0, new balance: -100.0
Withdraw: 100.0, new balance: -200.0
Deposit: 100.0, new balance: -100.0
Withdraw: 100.0, new balance: -200.0
Withdraw: 100.0, new balance: -100.0
Deposit: 100.0, new balance: 0.0
Withdraw: 100.0, new balance: -100.0
Withdraw: 100.0, new balance: 0.0
```

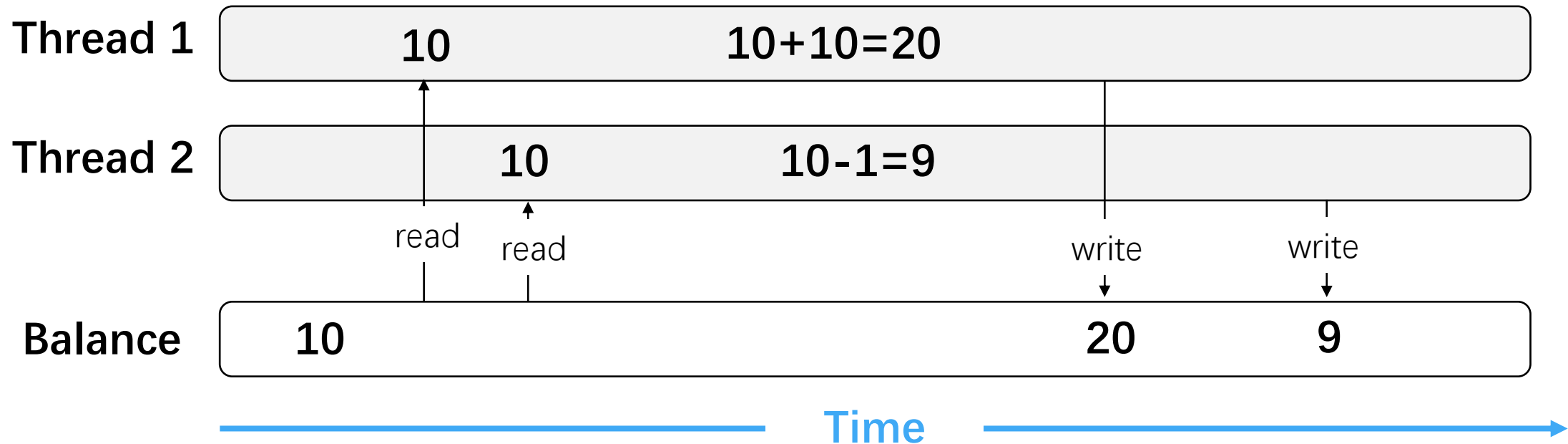


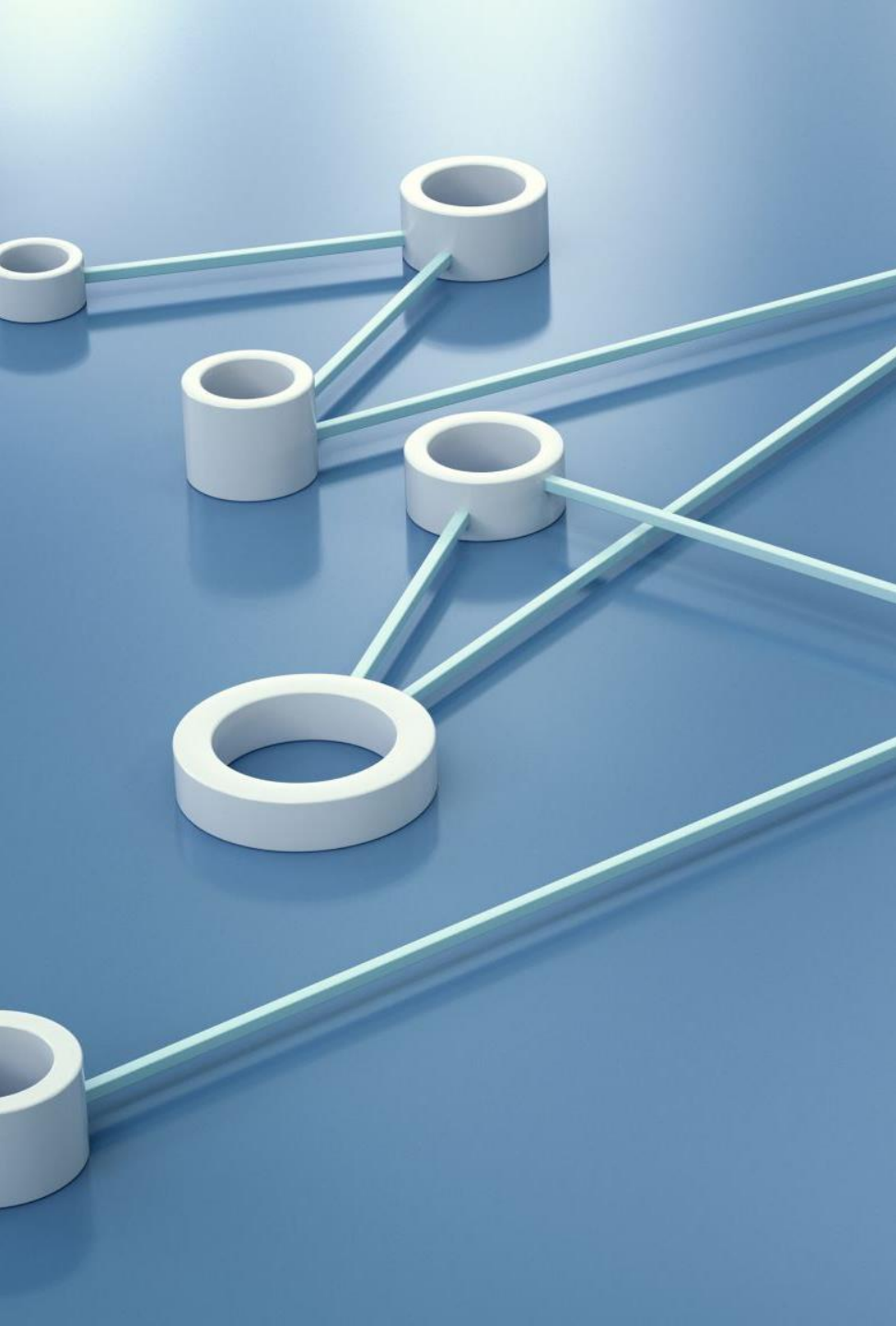
# Race Condition

- A concurrency problem/bug
- Multiple threads compete for a shared resource (race)
- The final results depend on which thread gets the resource first (**non-deterministic**)

# Critical Section

- The part of the program which accesses the shared resource
- A critical section is executed by multiple threads, and the sequence of execution for the threads makes a difference in the result





# Synchronization in Java

---

- The synchronization mechanism ensures that only one thread can access the critical section (shared resource) at a given time
- Java supports
  - The **Concurrency** API (`java.util.concurrent`), introduced in Java 5
  - The **synchronized** keyword in JDK 1.0

# Using Lock

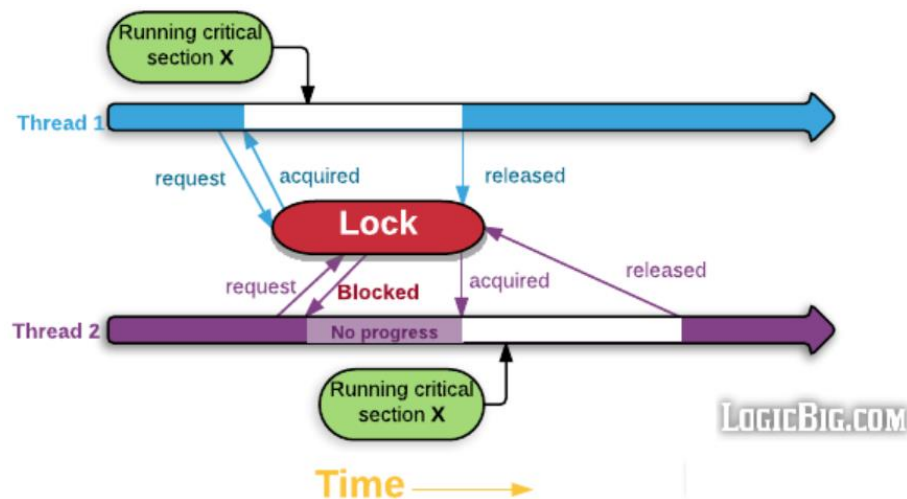
- Lock is used to control the threads that want to manipulate a shared resource
- Since Lock is an interface, we cannot create an instance of Lock directly; we should create an instance of a class that implements the Lock interface
- Java provides several implementations of Lock; ReentrantLock is the most used one

```
public class BankAccount {  
    private double balance;  
    private Lock balanceChangeLock;  
  
    public BankAccount(){  
        balance = 0;  
        balanceChangeLock = new ReentrantLock();  
    }  
  
    public void withdraw(double amount){  
        balanceChangeLock.lock();  
  
        balance -= amount;  
        System.out.printf("Withdraw: %.1f, " +  
            "new balance: %.1f\n", amount, balance);  
  
        balanceChangeLock.unlock();  
    }  
  
    public void deposit(double amount){  
        balanceChangeLock.lock();  
  
        balance += amount;  
        System.out.printf("Deposit: %.1f, " +  
            "new balance: %.1f\n", amount, balance);  
  
        balanceChangeLock.unlock();  
    }  
}
```



# Using Lock

- When the Lock instance is locked, any other thread calling lock() will be **blocked** until the thread that locked the lock calls unlock().
- When unlock() is called, the Lock is unlocked so other threads can lock it.



```
public class BankAccount {
    private double balance;
    private Lock balanceChangeLock;

    public BankAccount(){
        balance = 0;
        balanceChangeLock = new ReentrantLock();
    }

    public void withdraw(double amount){
        balanceChangeLock.lock();

        balance -= amount;
        System.out.printf("Withdraw: %.1f, " +
            "new balance: %.1f\n", amount, balance);

        balanceChangeLock.unlock();
    }

    public void deposit(double amount){
        balanceChangeLock.lock();

        balance += amount;
        System.out.printf("Deposit: %.1f, " +
            "new balance: %.1f\n", amount, balance);

        balanceChangeLock.unlock();
    }
}
```

# Using Lock

Deposit: 100.0, new balance: 100.0  
Withdraw: 100.0, new balance: 0.0  
Deposit: 100.0, new balance: 100.0  
Withdraw: 100.0, new balance: 0.0  
Withdraw: 100.0, new balance: -100.0  
Withdraw: 100.0, new balance: -200.0  
Deposit: 100.0, new balance: -100.0  
Withdraw: 100.0, new balance: -200.0

To disallow negative balance during withdraw,  
we can wait for other threads to deposit money

Withdraw: 100.0, new balance: -200.0  
Deposit: 100.0, new balance: -100.0  
Deposit: 100.0, new balance: 0.0  
Deposit: 100.0, new balance: 100.0  
Withdraw: 100.0, new balance: 0.0  
Deposit: 100.0, new balance: 100.0  
Withdraw: 100.0, new balance: 0.0  
Deposit: 100.0, new balance: 100.0  
Withdraw: 100.0, new balance: 0.0

```
public class BankAccount {
    private double balance;
    private Lock balanceChangeLock;

    public BankAccount(){
        balance = 0;
        balanceChangeLock = new ReentrantLock();
    }

    public void withdraw(double amount){
        balanceChangeLock.lock();

        balance -= amount;
        System.out.printf("Withdraw: %.1f, " +
            "new balance: %.1f\n", amount, balance);

        balanceChangeLock.unlock();
    }

    public void deposit(double amount){
        balanceChangeLock.lock();

        balance += amount;
        System.out.printf("Deposit: %.1f, " +
            "new balance: %.1f\n", amount, balance);

        balanceChangeLock.unlock();
    }
}
```

# Using Lock

```
Deposit: 100.0, new balance: 100.0
Withdraw: 100.0, new balance: 0.0
Deposit: 100.0, new balance: 100.0
Withdraw: 100.0, new balance: 0.0
Withdraw: 100.0, new balance: -100.0
Withdraw: 100.0, new balance: -200.0
Deposit: 100.0, new balance: -100.0
Withdraw: 100.0, new balance: -200.0
```

To disallow negative balance during withdraw,  
we can wait for other threads to deposit money

```
Withdraw: 100.0, new balance: -200.0
Deposit: 100.0, new balance: -100.0
Deposit: 100.0, new balance: 0.0
Deposit: 100.0, new balance: 100.0
Withdraw: 100.0, new balance: 0.0
Deposit: 100.0, new balance: 100.0
Withdraw: 100.0, new balance: 0.0
Deposit: 100.0, new balance: 100.0
Withdraw: 100.0, new balance: 0.0
```

```
public void withdraw(double amount)
{
    balanceChangeLock.lock();
    try
    {
        while (balance < amount)
        {
            Wait for the balance to grow.
        }
        . . .
    }
    finally
    {
        balanceChangeLock.unlock();
    }
}
```

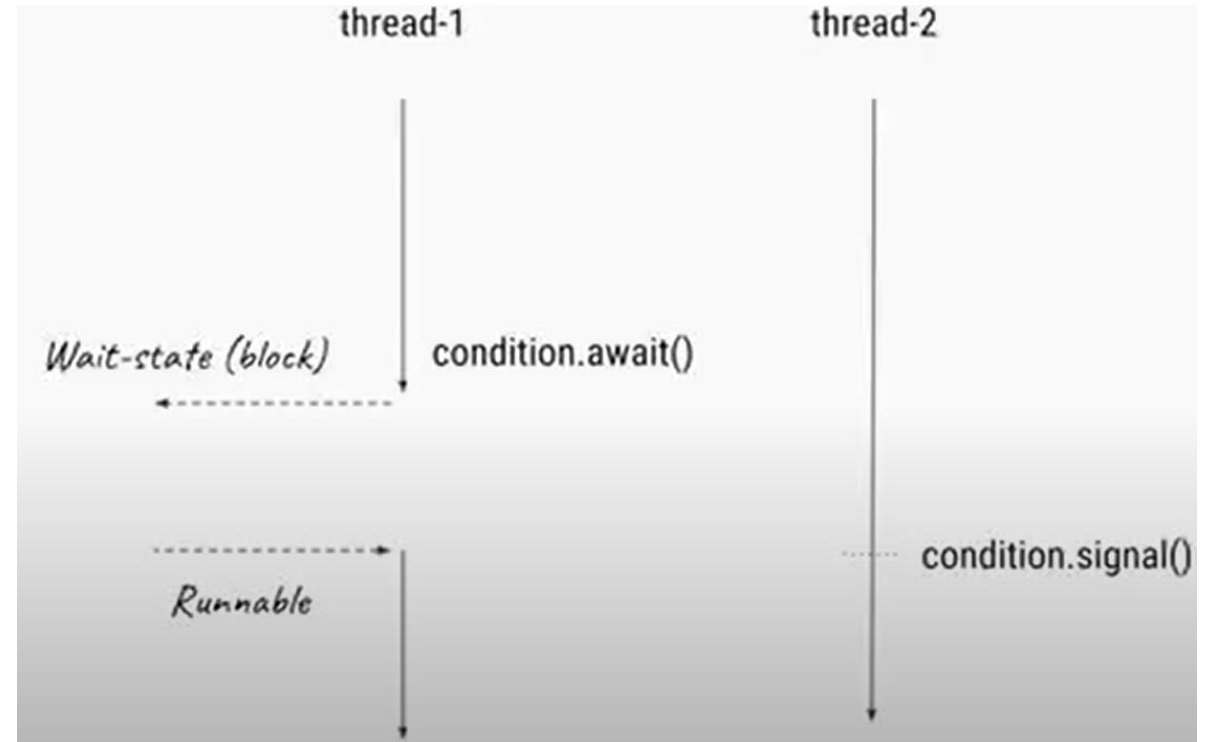
# Deadlock

- Other threads calling deposit() are blocked and waiting for withdraw() to unlock() the resource
- But withdraw() is waiting for deposit() to execute so that balance becomes enough for withdrawal.

```
public void withdraw(double amount)
{
    balanceChangeLock.lock();
    try
    {
        while (balance < amount)
        {
            Wait for the balance to grow.
        }
        . . .
    }
    finally
    {
        balanceChangeLock.unlock();
    }
}
```

# Avoiding Deadlocks

- The Condition interface (`java.util.concurrent.locks`) provides a thread ability to suspend its execution, until the given condition is true.
- Condition allows a thread
  - To temporarily release a lock so that another thread can proceed
  - To regain the lock later when the condition is satisfied



# Using Condition

- Each condition object belongs to a specific lock object.
- A lock can have one or more conditions.
- We could obtain a condition object with the `newCondition()` method of the `Lock` interface
- It is customary to give the condition object a name that describes the condition that you want to test

```
public class BankAccount {  
    private double balance;  
    private Lock balanceChangeLock;  
    private Condition sufficientBalanceCondition;  
  
    public BankAccount(){  
        balance = 0;  
        balanceChangeLock = new ReentrantLock();  
        sufficientBalanceCondition = balanceChangeLock.newCondition();  
    }  
}
```

# Using Condition

- For a condition to take effect, we need to implement an appropriate test (i.e., condition)
- For as long as the condition is not fulfilled, call the `await()` method on the condition object (hence the loop)
- Calling `await()` on a condition object makes the current thread deactivated and allows another thread to acquire the lock object.
- Current thread can proceed until some other thread invokes the `signalAll()` method on the same condition

```
public void withdraw(double amount) throws InterruptedException {  
    balanceChangeLock.lock();  
  
    try{  
        while(balance < amount){  
            sufficientBalanceCondition.await();  
        }  
        balance -= amount;  
        System.out.printf("Withdraw: %.1f, " +  
            "new balance: %.1f\n", amount, balance);  
    } finally {  
        balanceChangeLock.unlock();  
    }  
}
```

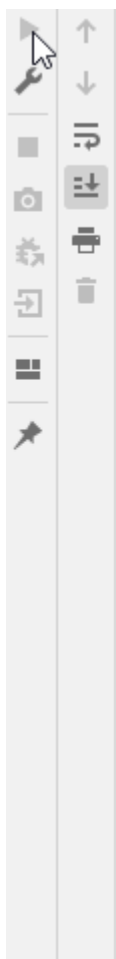


# Using Condition

- To unblock, another thread must execute the `signalAll()` method on the same condition object
- The `signalAll()` method unblocks all threads waiting on the condition, which then compete with each other that is waiting for the lock object.
- Eventually, one of them will gain access to the lock, and it will exit from the `await()` method.

```
public void deposit(double amount){  
    balanceChangeLock.lock();  
  
    try{  
        balance += amount;  
        System.out.printf("Deposit: %.1f, " +  
            "new balance: %.1f\n", amount, balance);  
  
        sufficientBalanceCondition.signalAll();  
    } finally {  
        balanceChangeLock.unlock();  
    }  
}
```

# Synchronized Bank Transfer



```
public class BankTransfer {  
    public static void main(String[] args) {  
        BankAccount account = new BankAccount();  
  
        Runnable depositActivity = () -> {  
            account.deposit( amount: 100);  
        };  
  
        Runnable withdrawActivity = () -> {  
            account.withdraw( amount: 100);  
        };  
  
        for(int threads = 1; threads <= 10; threads++){  
            Thread depositThread = new Thread(depositActivity);  
            Thread withdrawThread = new Thread(withdrawActivity);  
  
            depositThread.start();  
            withdrawThread.start();  
        }  
    }  
}
```

# The synchronized Keyword

- Since JDK 1.0, every object in Java has an intrinsic lock (内在锁)
- If a method is declared with the synchronized keyword, to call the method, a thread must acquire the intrinsic lock of the object owning the method

```
public synchronized void method()  
{  
    method body  
}
```

=

```
public void method()  
{  
    this.intrinsicLock.lock();  
    try  
    {  
        method body  
    }  
    finally { this.intrinsicLock.unlock(); }  
}
```

# The synchronized Keyword

- Since JDK 1.0, every object in Java has an intrinsic lock (内在锁)
- If a method is declared with the synchronized keyword, to call the method, a thread must acquire the intrinsic lock of the object owning the method
- The intrinsic object lock has a single associated condition
- We can call `wait()` to wait for this condition, and `notify()/notifyAll()` to unblock waiting threads

# synchronized methods

- A synchronized instance method is synchronized on the instance (object) owning the method
- When one thread is executing a synchronized method for an object, all other threads that invoke any synchronized methods for the same object block until the first thread is done with the object.

```
public class BankAccount {
    private double balance;

    public BankAccount(){
        balance = 0;
    }

    public synchronized void withdraw(double amount)
        throws InterruptedException {

        while(balance < amount){
            wait();
        }
        balance -= amount;
        System.out.printf("Withdraw: %.1f, " +
            "new balance: %.1f\n", amount, balance);
    }

    public synchronized void deposit(double amount){
        balance += amount;
        System.out.printf("Deposit: %.1f, " +
            "new balance: %.1f\n", amount, balance);

        notifyAll();
    }
}
```

# synchronized block

- Synchronized block synchronizes only part of the method
- A synchronized block takes an object in parentheses, which is called a monitor object.
- Only one thread can execute inside a Java code block synchronized on the same monitor object.

```
public class BankAccount {
    private double balance;

    public BankAccount() {
        balance = 0;
    }

    public void withdraw(double amount)
        throws InterruptedException {
        synchronized (this) {
            while (balance < amount) {
                wait();
            }
            balance -= amount;
            System.out.printf("Withdraw: %.1f, " +
                              "new balance: %.1f\n", amount, balance);
        }
    }

    public void deposit(double amount) {
        synchronized (this) {
            balance += amount;
            System.out.printf("Deposit: %.1f, " +
                              "new balance: %.1f\n", amount, balance);

            notifyAll();
        }
    }
}
```

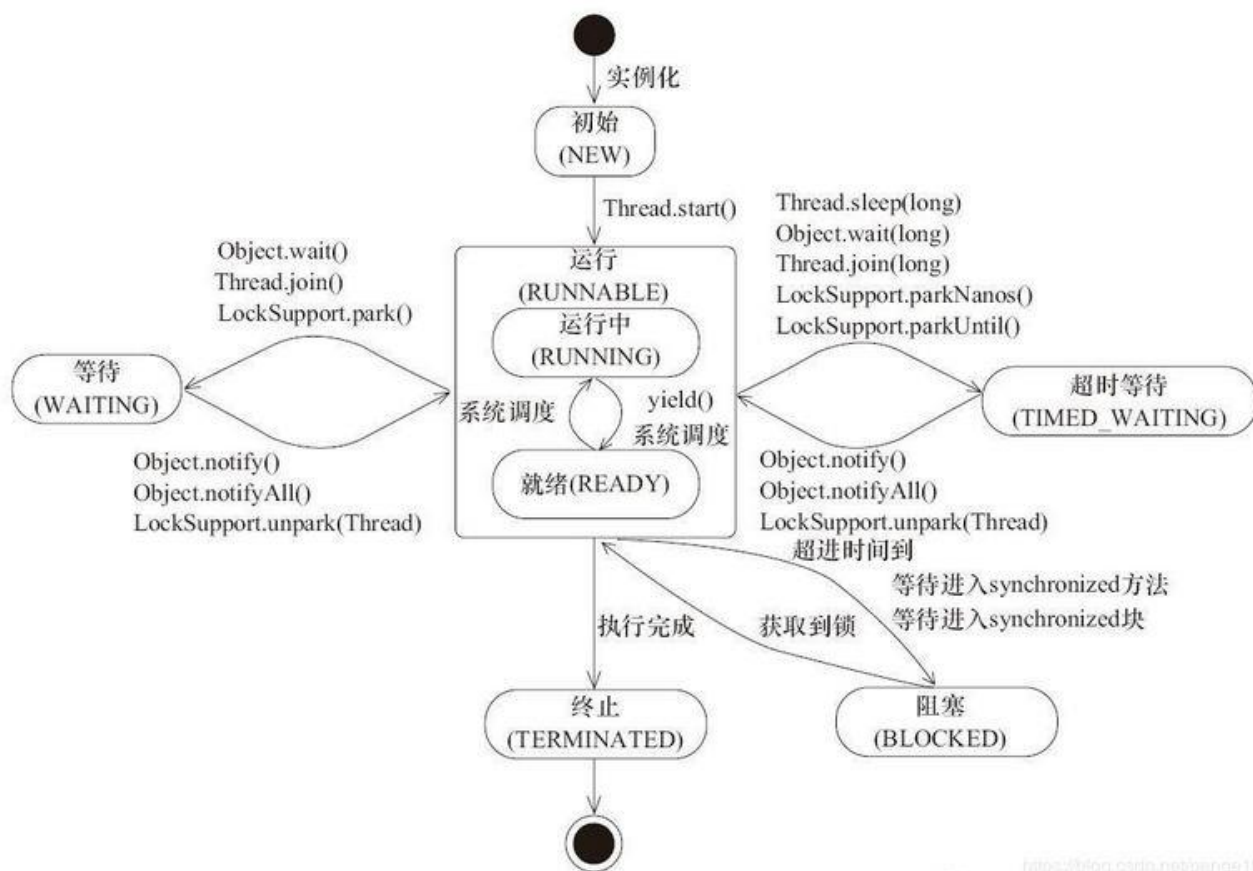
# synchronized block

- c1 and c2 are independent variables, which can be updated concurrently
- Synchronizing inc1() and inc2() creates unnecessary blocking. Instead, synchronizing blocks improves concurrency

```
public class SyncDemo {  
  
    private long c1 = 0;  
    private long c2 = 0;  
  
    private Object lock1 = new Object();  
    private Object lock2 = new Object();  
  
    public void inc1() {  
        // 20 lines of code omitted...  
        synchronized(lock1) {  
            c1++;  
        }  
    }  
  
    public void inc2() {  
        // 30 lines of code omitted...  
        synchronized(lock2) {  
            c2++;  
        }  
    }  
}
```



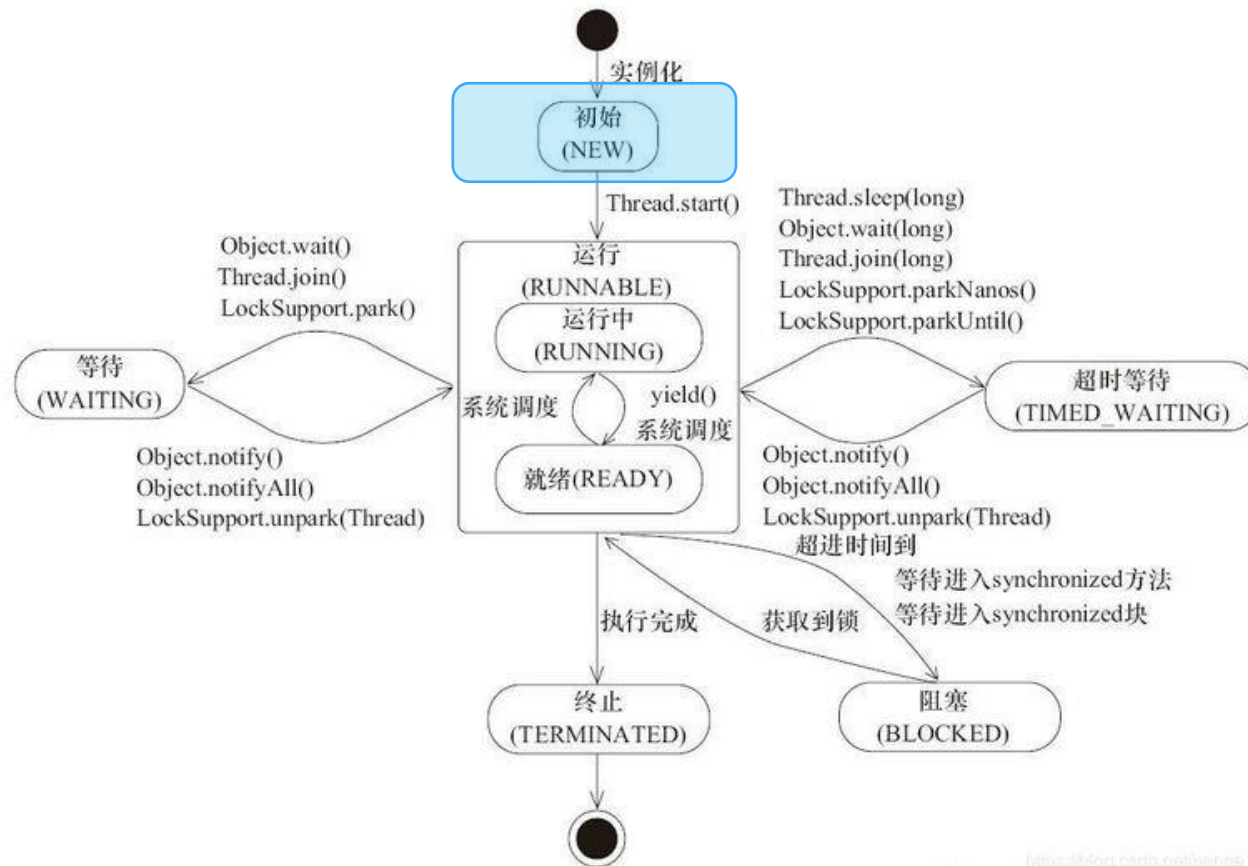
# Thread States



A thread can be in one of the following states (Enum Thread.State):

- NEW
- RUNNABLE
- BLOCKED
- WAITING
- TIMED\_WAITING
- TERMINATED

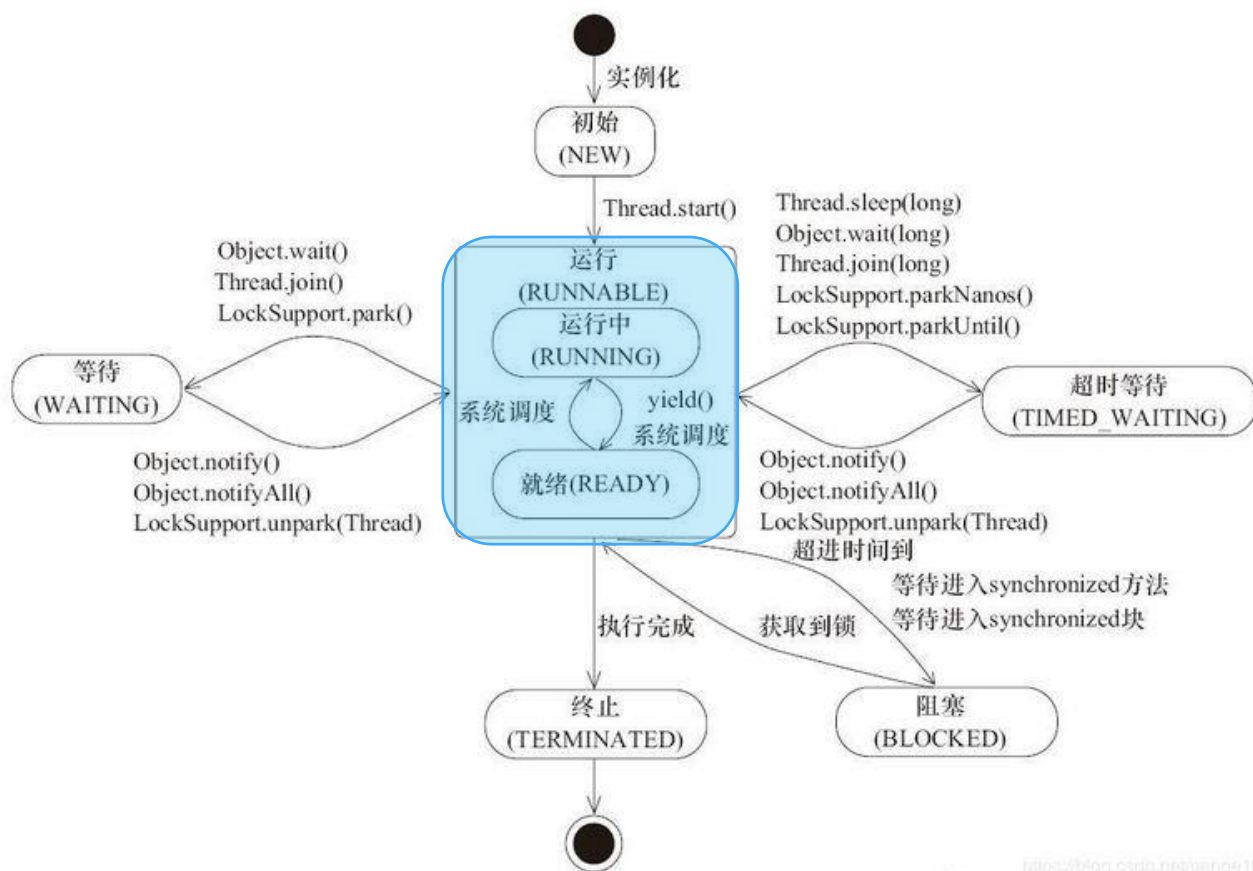
# Thread States



## NEW

- When you create a thread with new (e.g., `new Thread(r)`), it enters this initial NEW state
- At this state, the program has NOT started executing code

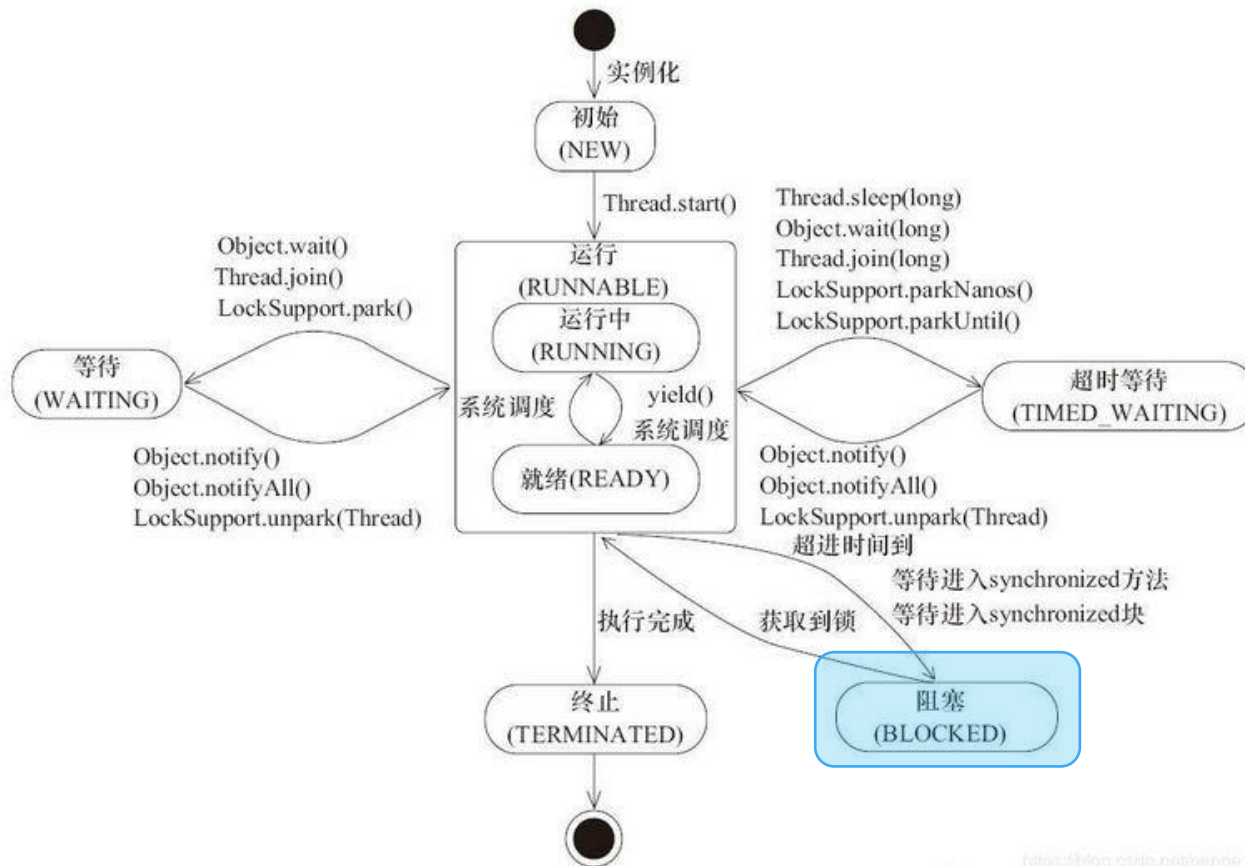
# Thread States



## RUNNABLE

- Ready to run (`Thread.start()`)
- Nothing prevents the thread from “running” except the availability of a CPU to run on

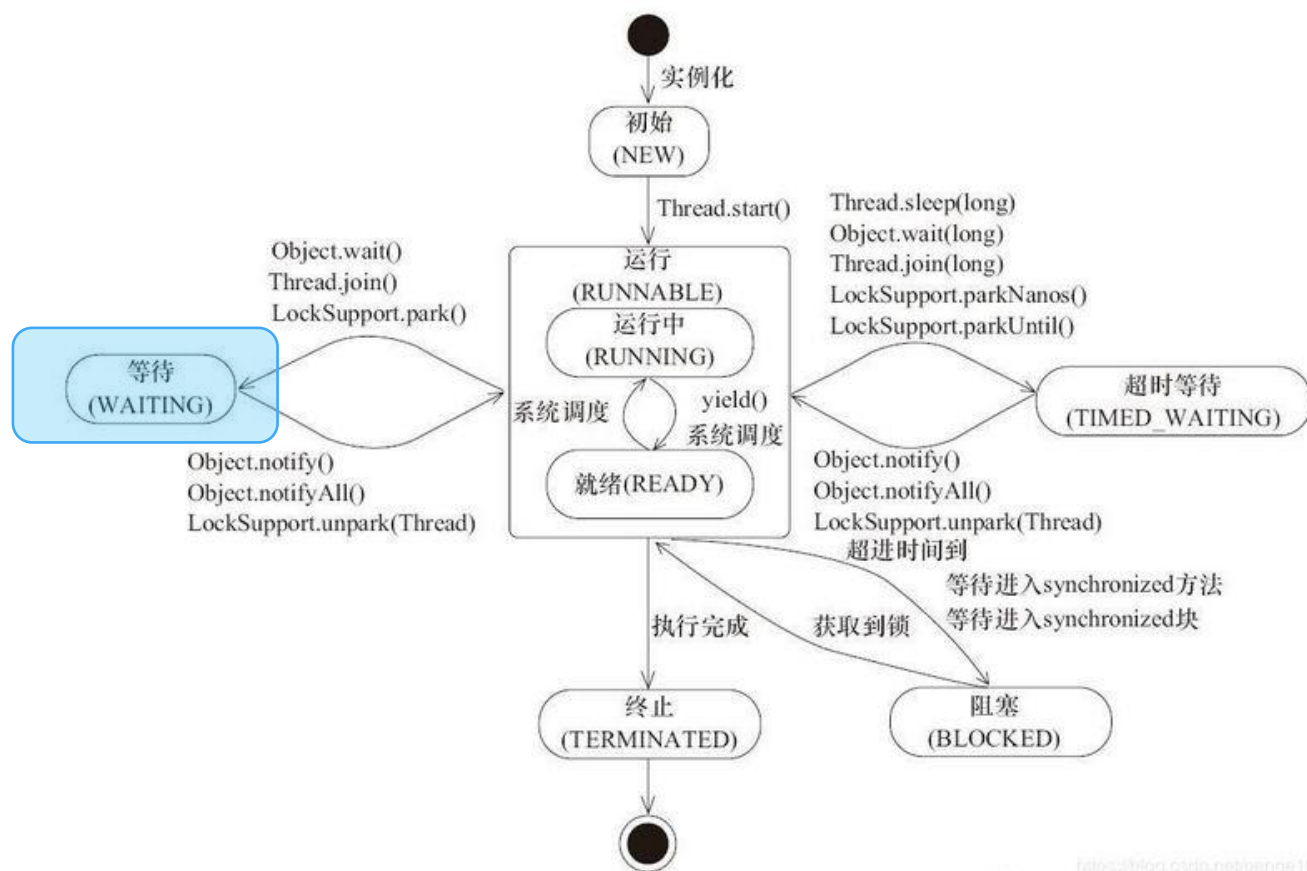
# Thread States



## BLOCKED

- When a thread tries to acquire an intrinsic object lock (synchronized keyword) that is currently held by another thread, it becomes blocked.
- The thread is unblocked when all other threads have released the lock

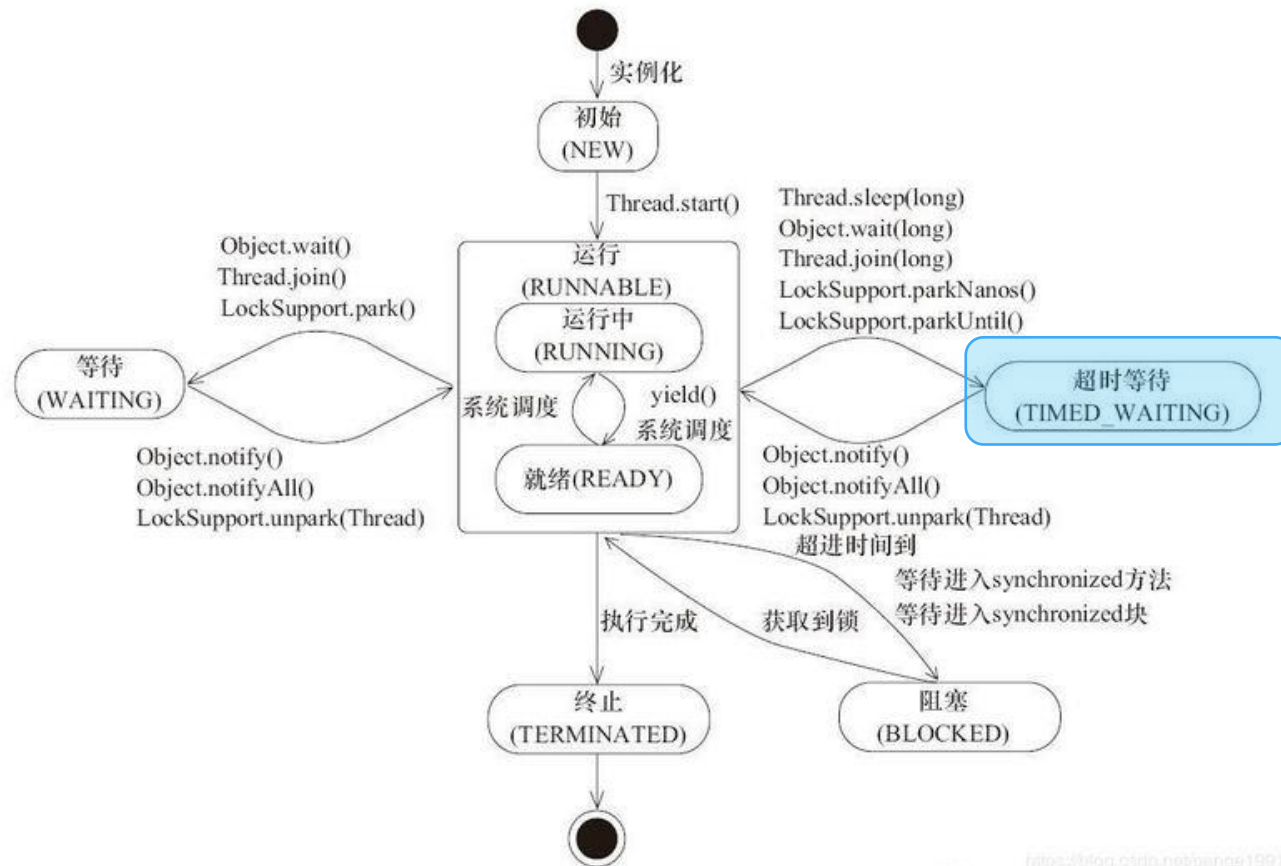
# Thread States



## WAITING

- In the WAITING state, a thread is waiting for a signal from another thread.
- This happens typically by calling `Object.wait()`, or `Thread.join()`.
- The thread will then remain in this state until another thread calls `Object.notify()`, `Object.notifyAll()`, or dies.

# Thread States

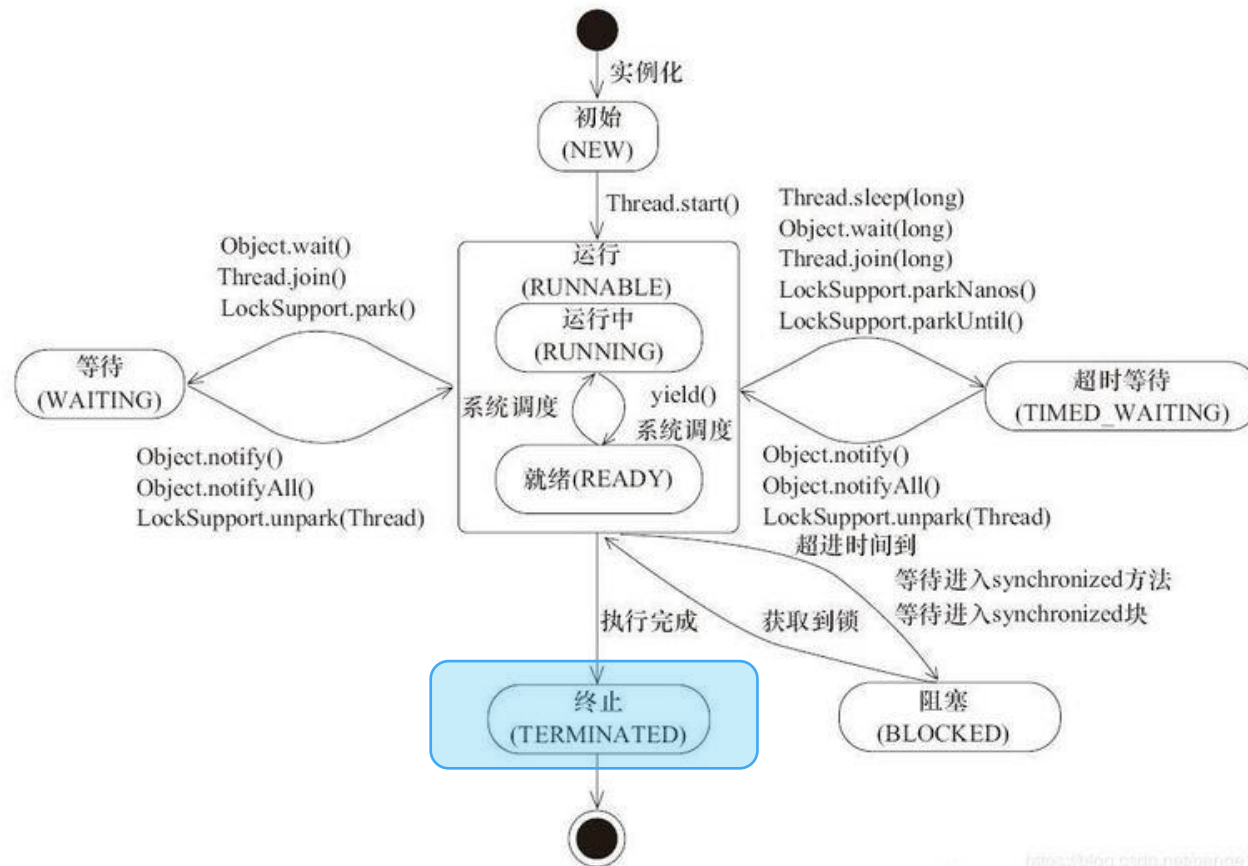


## TIMED\_WAITING

- Several methods support timeout
- Calling them causes the thread to enter TIMED\_WAITING state



# Thread States



## TERMINATED

- The run() method exits normally
- The run() method dies abruptly because of an uncaught exception



# The volatile Keyword

- **synchronized** keyword is **heavyweight**: the entire object is blocked
- Lock can be used, but requires extra code

```
private boolean done;  
public synchronized boolean isDone() { return done; }  
public synchronized void setDone() { done = true; }
```

- The **volatile** keyword offers a **lock-free** mechanism for synchronizing access to an instance field, ensuring that **a change to the **volatile** variable in one thread is **visible** from any other thread that reads the variable.**

```
private volatile boolean done;  
public boolean isDone() { return done; }  
public void setDone() { done = true; }
```

# The volatile Keyword - Visibility

- The compiler will insert the appropriate code to ensure **visibility**
  - For write (assignment) operation, insert an additional **store** instruction to force the thread to immediately write the new value back to main memory (for non-volatile variables, the new value may be write back to main memory later before the thread ends)
  - For read operation, insert an additional **load** instruction to force the thread to read the value from the main memory rather than from the working memory

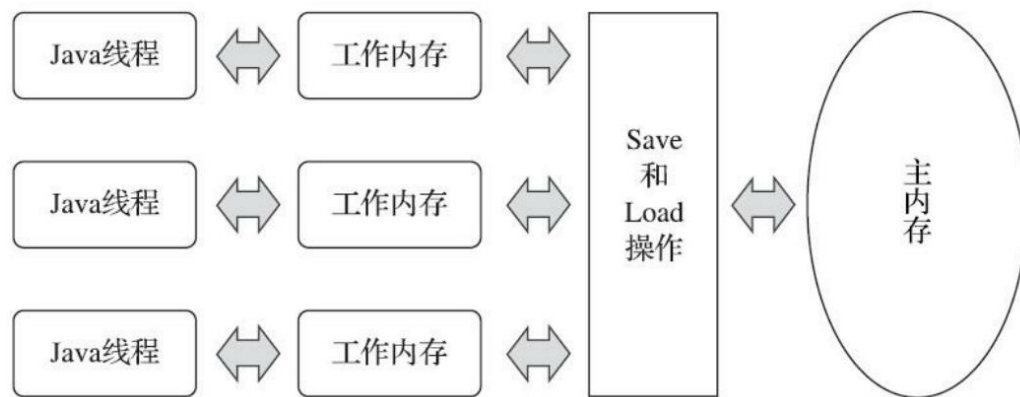


Image source: 《深入理解Java虚拟机》

```
private volatile boolean done;  
public boolean isDone() { return done; }  
public void setDone() { done = true; }
```

# The volatile Keyword Atomicity

- In programming, an *atomic* action is one that effectively **happens all at once**. An atomic action **cannot stop in the middle**: it either happens completely, or it doesn't happen at all.
- The **volatile** variables do NOT provide atomicity.
- **race++**: **race = race + 1**
  - (Read) Read the latest **race** value
  - Create constant 1
  - Add 1 to race
  - (Write) Assign the new value to **race**
- During which other threads may have altered **race**

```
public class VolatileTest {
    public static volatile int race = 0;

    public static void increase() {
        race++;
    }

    private static final int THREADS_COUNT = 20;

    public static void main(String[] args) {
        Thread[] threads = new Thread[THREADS_COUNT];
        for (int i = 0; i < THREADS_COUNT; i++) {
            threads[i] = new Thread(() -> {
                for(int j=0; j<1000;j++) increase();
            });
            threads[i].start();
        }

        // Wait for all other threads to finish
        try {
            for (Thread thread : threads) {
                thread.join();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // expected: race = 20000
        System.out.println(race);
    }
}
```

# The volatile Keyword

A better scenario to use **volatile**:  
Perform no other operations than atomic assignment and read

```
volatile boolean shutdownRequested;  
  
public void shutdown() {  
    shutdownRequested = true;  
}  
  
public void doWork() {  
    while (!shutdownRequested) {  
        // 代码的业务逻辑  
    }  
}
```

# Atomics

- A number of classes in the `java.util.concurrent.atomic` package use efficient machine-level instructions to guarantee atomicity of other operations **without using locks**.
- For example, `AtomicInteger` has methods `incrementAndGet()` and `decrementAndGet()` that atomically increment or decrement an integer.

```
public class AtomicTest {
    public static AtomicInteger race =
        new AtomicInteger( initialValue: 0);

    public static void increase() {
        race.incrementAndGet();
    }

    private static final int THREADS_COUNT = 20;

    public static void main(String[] args) {
        Thread[] threads = new Thread[THREADS_COUNT];
        for (int i = 0; i < THREADS_COUNT; i++) {
            threads[i] = new Thread(() -> {
                for(int j=0; j<1000;j++) increase();});
            threads[i].start();
        }

        // Wait for all other threads to finish
        try {
            for (Thread thread : threads) {
                thread.join();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // expected: race = 20000
        System.out.println(race);
    }
}
```

# CAS (Compare-And-Swap)

1. Read the variable value (O) from memory (M)
2. Calculate the new value (N)
3. Check if O is equal to the value in M. if so, write N to M; otherwise, retry (some other threads change the value in M)

```
/**
 * Atomically increment by one the current value.
 * @return the updated value
 */
public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}
```

Guarantee the correct value is computed and returned, even if multiple threads access the same instance concurrently





# Lecture 7

---

- Overview
- Creating & Starting Threads
- Synchronization
- Thread-safe Collections
- Tasks and Thread Pools

# Concurrency for Java Collection

- All collection classes (e.g., ArrayList, HashMap, HashSet, TreeSet, etc.) in `java.util` are not thread-safe (except for Vector and Hashtable). Why?
- Synchronization can be expensive
  - Vector and Hashtable are the two collections exist early and are designed for thread-safety from the start. However, they quickly expose poor performance
  - New collections (List, Set, Map, etc) provide no concurrency control to provide maximum performance in single-threaded applications

<https://www.codejava.net/java-core/collections/understanding-collections-and-thread-safety-in-java>



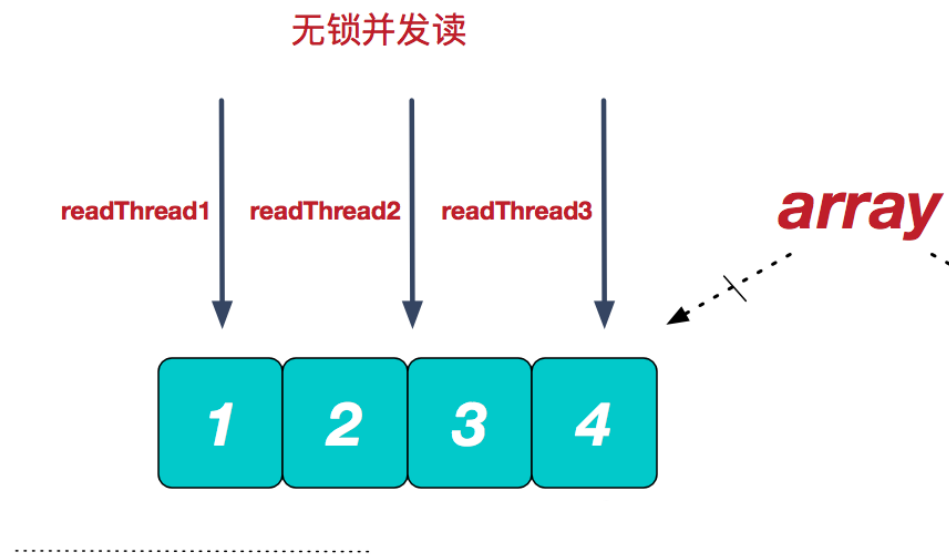
# Concurrent Collections in Java

- Introduced in Java 5 in `java.util.concurrent` package
- 3 categories w.r.t. thread-safety mechanism
  - Copy-on-Write collections
  - Compare-and-Swap collections (CAS)
  - Collections using Lock



# CopyOnWriteArrayList

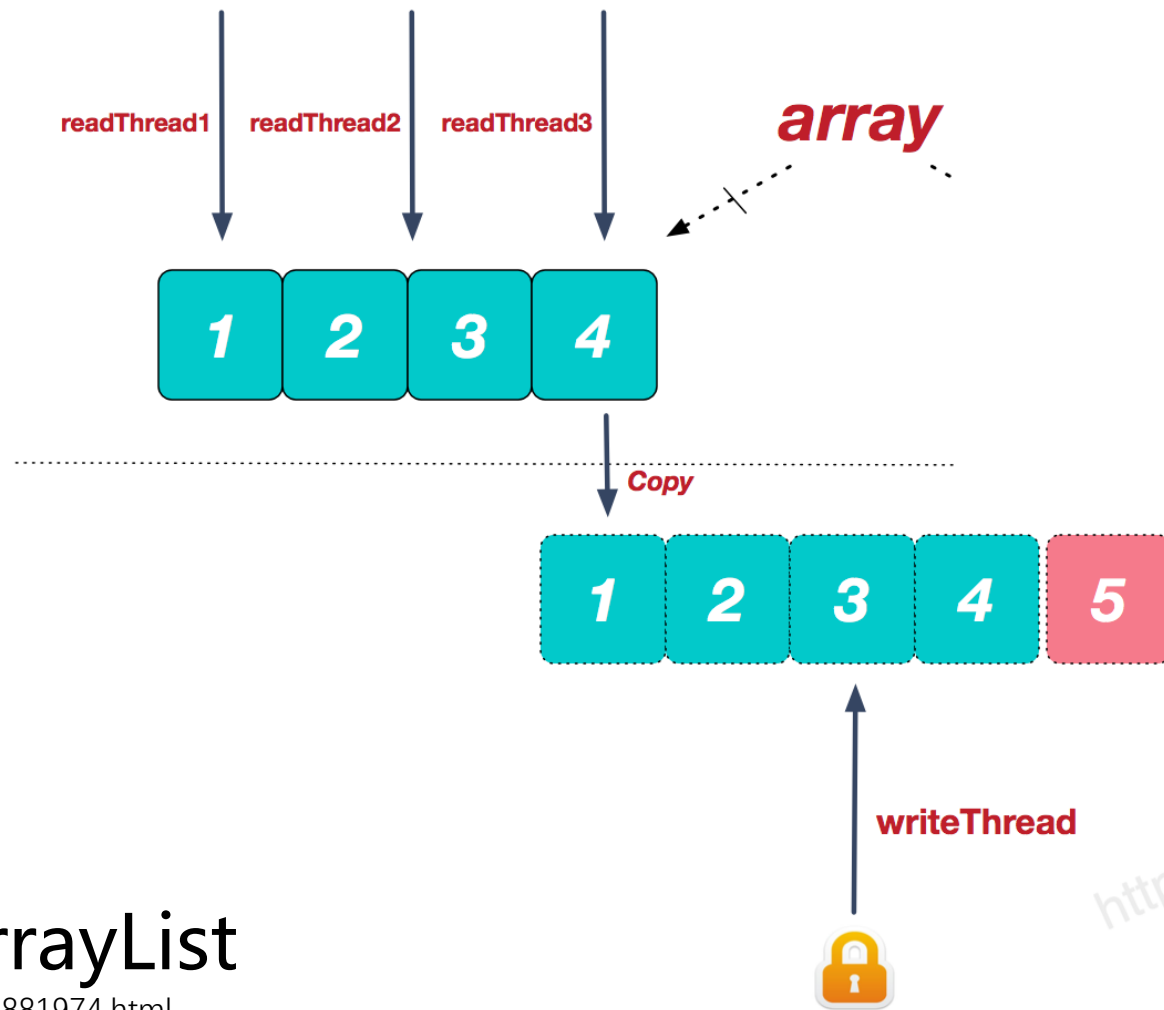
- CopyOnWriteArrayList implements the List interface (i.e., it has all typical behaviors of a List)
- CopyOnWriteArrayList is considered as a **thread-safe alternative** to ArrayList



# CopyOnWriteArrayList

<https://www.cnblogs.com/chengxiao/p/6881974.html>

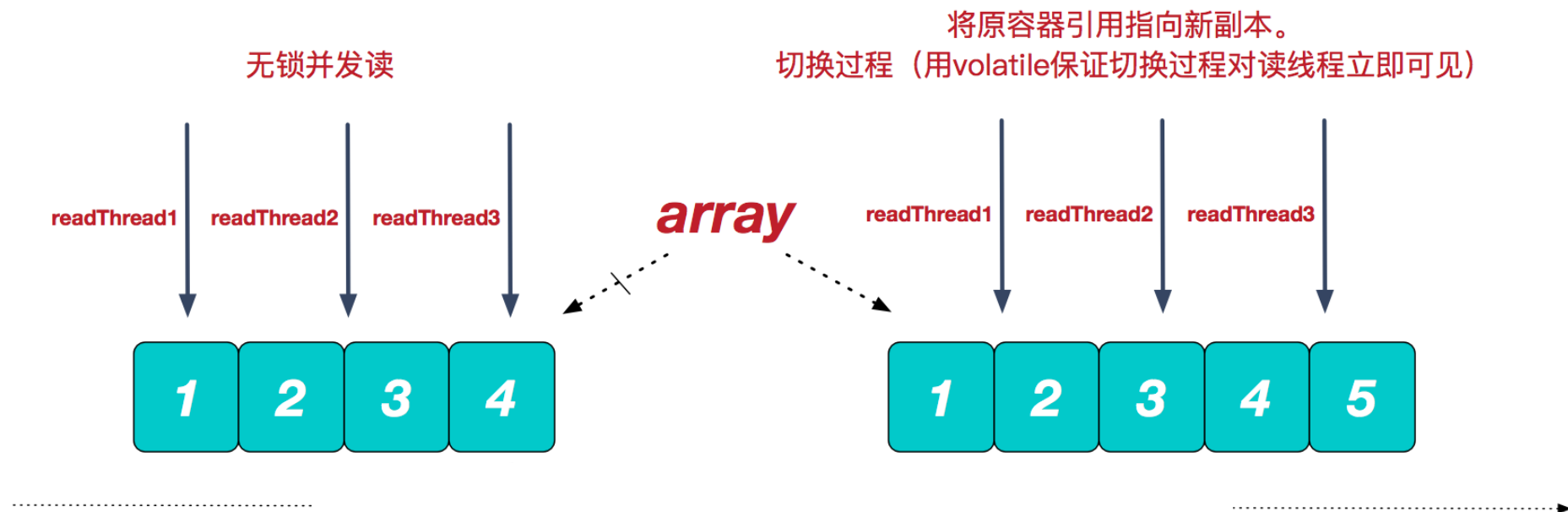
无锁并发读



# CopyOnWriteArrayList

<https://www.cnblogs.com/chengxiao/p/6881974.html>

将原容器拷贝一份，写操作则作用在新副本上，需加锁。  
此过程中若有读操作则会作用在原容器上



# CopyOnWriteArrayList

<https://www.cnblogs.com/chengxiao/p/6881974.html>



# CopyOnWriteArrayList

Under the hood: copy-on-write collections store values in an immutable array; any change to the value of the collection results in a new array being created to reflect the new values

```
public boolean add(E e) {  
    final ReentrantLock lock = this.lock;  
    lock.lock();  
    try {  
        Object[] elements = getArray();  
        int len = elements.length;  
        Object[] newElements = Arrays.copyOf(elements, len + 1);  
        newElements[len] = e;  
        setArray(newElements);  
        return true;  
    } finally {  
        lock.unlock();  
    }  
}
```

# Copy-on-Write Collections

- Behaviors: sequential writes and concurrent reads
  - Reads do not block
  - Writes do not block reads, but only one write can occur at once
- Better used in applications in which reads >>>> writes
- Example classes
  - CopyOnWriteArrayList
  - CopyOnWriteArraySet

```

List<Integer> arrayList = new ArrayList<>();
List<Integer> copyOnWriteArrayList = new CopyOnWriteArrayList<>();

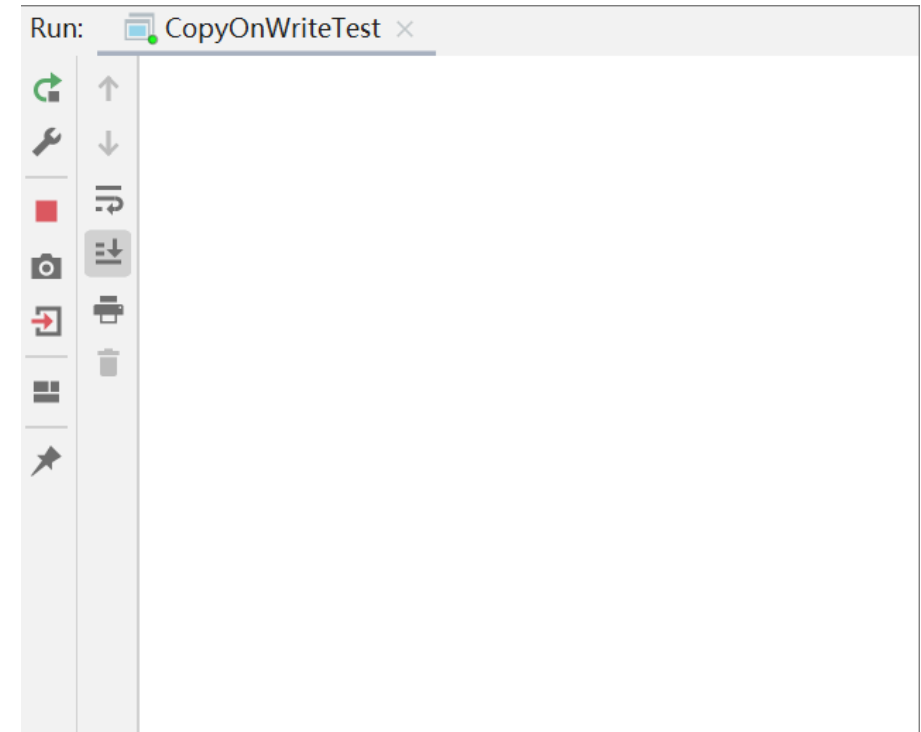
Thread thread1 = new Thread(() -> {
    for (int i = 0; i < 5; i++) {
        arrayList.add(i);
        copyOnWriteArrayList.add(i);
        try {Thread.sleep(100);}
        } catch (InterruptedException e) {e.printStackTrace();}
    }
});

Thread thread2 = new Thread(() -> {
    for (int i = 0; i < 5; i++) {
        arrayList.add(i);
        copyOnWriteArrayList.add(i);
        try {Thread.sleep(100);}
        } catch (InterruptedException e) {e.printStackTrace(); }
    }
});

thread1.start();
thread2.start();

thread1.join(); thread2.join();


```



```

System.out.println("ArrayList size: " + arrayList.size());
System.out.println("CopyOnWriteArrayList size: " + copyOnWriteArrayList.size());

```




## Fail-fast Iterator

```
List<Integer> list = new ArrayList<>(List.of(1,2,3,4));
Iterator<Integer> itr = list.iterator();
while (itr.hasNext()) {
    Integer no = itr.next();
    System.out.println(no);
    if (no == 3)
        // ConcurrentModificationException
        list.add(5);
}
```

1  
2  
3

```
Exception in thread "main" java.util.ConcurrentModificationException Create breakpoint
    at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1013)
    at java.base/java.util.ArrayList$Itr.next(ArrayList.java:967)
    at multithreading.FailSafeExample.main(FailSafeExample.java:28)
```



## Fail-safe Iterator

The "snapshot" style iterator method uses a reference to the state of the array at the point that the iterator was created.

```
CopyOnWriteArrayList<Integer> list = new
    CopyOnWriteArrayList<>(new Integer[] { 1, 2, 3, 4 });

Iterator<Integer> itr = list.iterator();
while (itr.hasNext()) {
    Integer no = itr.next();
    System.out.println(no);
    if (no == 3)
        // No exception since it has created a separate copy
        // Yet list doesn't grow since itr is
        // on the snapshot of the old copy
        list.add(5);
}
System.out.println(list);
```

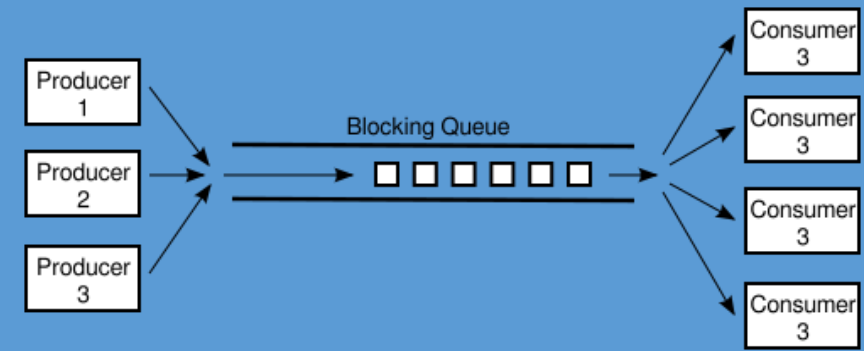
1  
2  
3  
4  
[1, 2, 3, 4, 5]



# Collections using Lock

- This mechanism divides the collection into parts that can be separately locked, giving improved concurrency
- Example classes
  - Most implementations of `BlockingQueue`
  - `ConcurrentHashMap`

# BlockingQueue



- A blocking queue causes a thread to block when
  - Adding an element to a queue that is full
  - Removing an element when the queue is empty
- Blocking queues are useful for coordinating work of multiple threads
  - Producer threads can periodically deposit intermediate results into a blocking queue
  - Consumers threads can remove the intermediate results and process them further
- Blocking queues automatically balances the workload
  - If producers run slower than consumers, consumers block while waiting for the results
  - If producers run faster, the queue blocks until consumers catch up

Image resources:

<https://math.hws.edu/eck/cs124/javanotes7/c12/s3.html>



```
BlockingQueue<Integer> queue = new LinkedBlockingQueue<>(5);
```

```
Runnable producer = ()->{  
    while(true){  
        try {  
            queue.put(1);  
            System.out.println("Put 1, " + queue);  
            Thread.sleep(100);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
};
```

```
Runnable consumer = ()->{  
    while(true){  
        try {  
            queue.take();  
            System.out.println("Take 1, " + queue);  
            Thread.sleep(100);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
};
```

```
new Thread(producer).start();  
new Thread(consumer).start();
```

```
Take 1, []  
Put 1, [1]  
Put 1, [1, 1]  
Put 1, [1, 1, 1]  
Take 1, [1, 1]  
Put 1, [1, 1, 1]  
Take 1, [1, 1]  
Put 1, [1, 1, 1]  
Put 1, [1, 1, 1, 1]  
Take 1, [1, 1, 1]  
Put 1, [1, 1, 1, 1]  
Take 1, [1, 1]  
Put 1, [1, 1, 1]  
Put 1, [1, 1, 1, 1]  
Put 1, [1, 1, 1, 1, 1]  
Take 1, [1, 1, 1, 1]
```

Coordinate smoothly within  
queue capacity

Summary of BlockingQueue methods

	<i>Throws exception</i>	<i>Special value</i>	<i>Blocks</i>	<i>Times out</i>
<b>Insert</b>	<code>add(e)</code>	<code>offer(e)</code>	<code>put(e)</code>	<code>offer(e, time, unit)</code>
<b>Remove</b>	<code>remove()</code>	<code>poll()</code>	<code>take()</code>	<code>poll(time, unit)</code>
<b>Examine</b>	<code>element()</code>	<code>peek()</code>	<i>not applicable</i>	<i>not applicable</i>

### 1. Throws Exception:

If the attempted operation is not possible immediately, an exception is thrown.

### 2. Special Value:

If the attempted operation is not possible immediately, a special value is returned (often true / false).

### 3. Blocks:

If the attempted operation is not possible immediately, the method call blocks until it is.

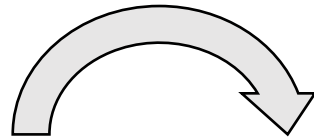
### 4. Times Out:

If the attempted operation is not possible immediately, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).

# Compare-And-Swap (CAS) Collections

- CAS: a technique used when designing concurrent algorithms

1. Make a local copy of the variable value (old value)



2. Calculate the new value

CAS (variable address, old value, new value)

3. Check if variable equals to the old value. if so, set variable to the new value; otherwise, retry (i.e., the variable must have been changed by another thread)

- Example classes: ConcurrentLinkedQueue, ConcurrentSkipListMap

<https://github.com/jaymoid/JavaConcurrentCollectionsDiagram>

©James Pittendreigh @jaymoid







# Lecture 7

---

- Overview
- Creating & Starting Threads
- Synchronization
- Thread-safe Collections
- **Tasks and Thread Pools**

# Why Use Thread Pools?

- Constructing a new thread is expensive
- If your program creates a large number of short-lived threads, you should not map each task to a separate thread, but use a thread pool instead.
- A thread pool contains a number of threads that are ready to run.
- You can focus on implementing the task, then simply give the task to the pool

**Decouple task implementation (business logic) and task execution**

# Tasks

- A `Runnable` is an asynchronous method with no parameters and no return value; it encapsulates a task that runs asynchronously
- A `Callable` is similar to a `Runnable`, but it returns a value

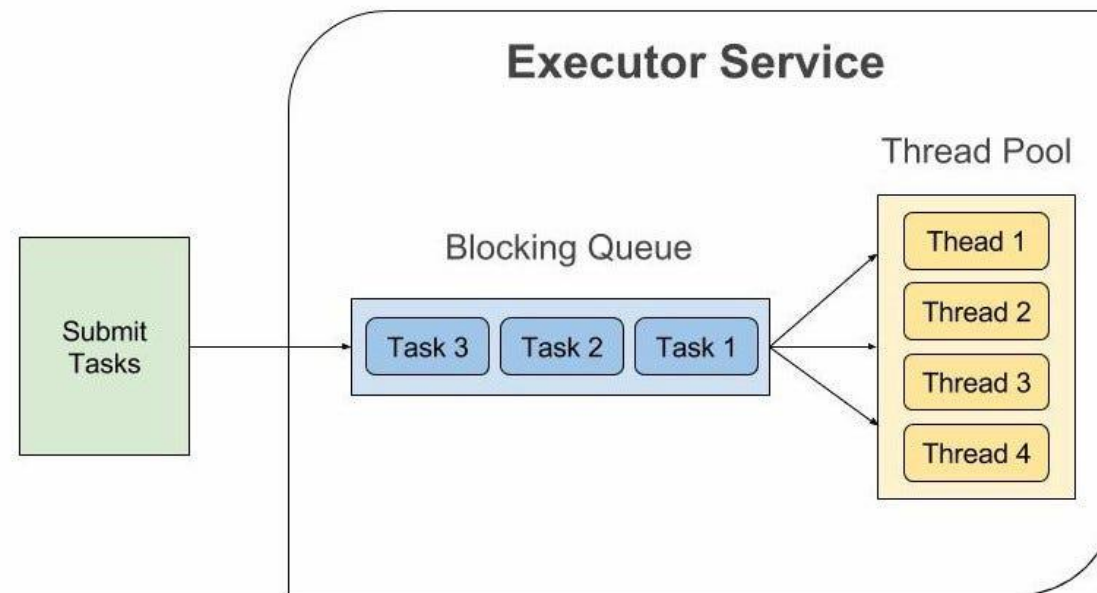
```
public interface Callable<V>
{
    V call() throws Exception;
}
```

e.g., a `Callable<Integer>` represents an asynchronous computation that eventually returns an `Integer` object.



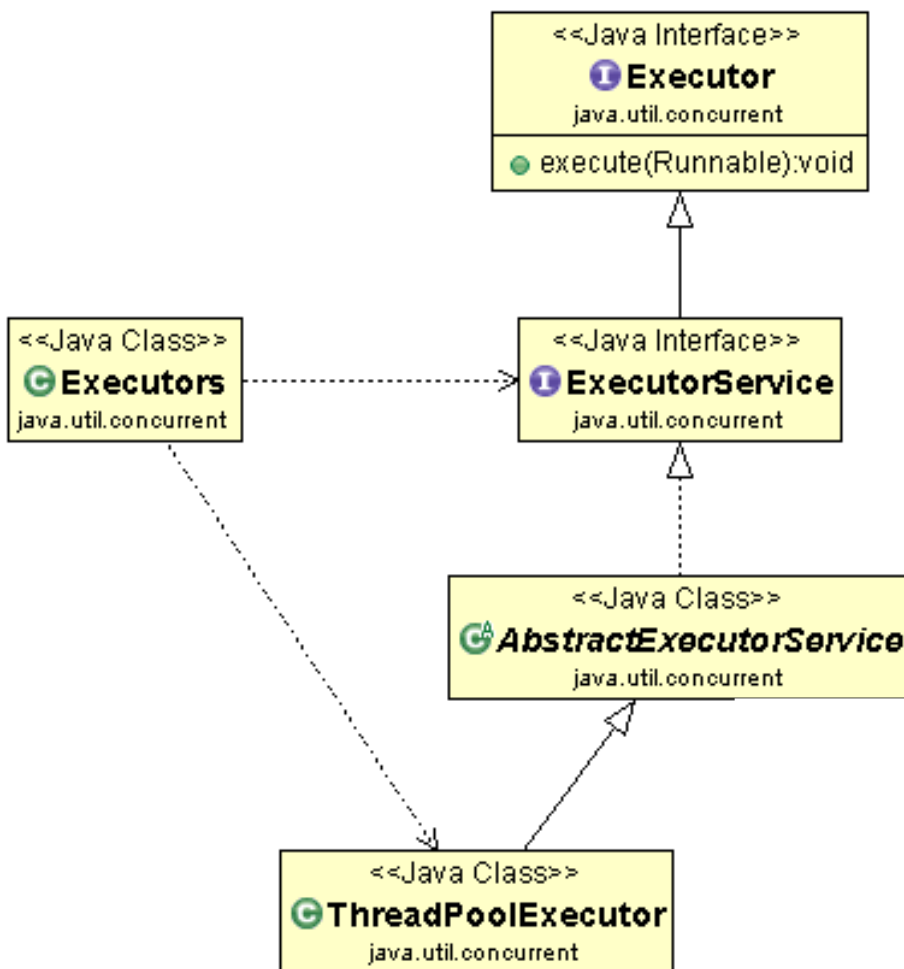
# Task Execution

The `ExecutorService` interface (`java.util.concurrent.ExecutorService`) represents an asynchronous execution mechanism which is capable of executing tasks concurrently in the background.



# Task Execution

- ThreadPoolExecutor is an implementation of ExecutorService and provides a pool of threads that executes the runnable or callable tasks.
- The Executors class has a number of static factory methods for constructing ThreadPoolExecutor



# Commonly Used Thread Pools

- **FixedThreadPool**: a thread pool with a fixed size. If more tasks are submitted than there are idle threads, the unserved tasks are placed on a queue. They are run when other tasks have completed.

```
ExecutorService executorService = Executors.newFixedThreadPool(3);
```

- **SingleThreadExecutor**: a degenerate pool of size 1 where a single thread executes the submitted tasks, one after another

```
ExecutorService executorService = Executors.newSingleThreadExecutor();
```

- **CachedThreadPool**: a thread pool that executes each task immediately, using an existing idle thread when available and creating a new thread otherwise.

```
ExecutorService executorService = Executors.newCachedThreadPool();
```

# Submitting Tasks

- You can submit a Runnable or Callable to an ExecutorService
- The pool will run the submitted task at its earliest convenience

```
Future<T> submit(Callable<T> task)  
Future<?> submit(Runnable task)
```

# Submitting Tasks

```
ExecutorService es1 = Executors.newFixedThreadPool( nThreads: 4);  
for (int i = 0; i < 6; i++) {  
    es1.submit(() -> System.out.println(Thread.currentThread().getName()  
        + " processing the task"));  
}  
es1.shutdown();
```

pool-1-thread-2 processing the task  
pool-1-thread-4 processing the task  
pool-1-thread-1 processing the task  
pool-1-thread-4 processing the task  
pool-1-thread-2 processing the task  
pool-1-thread-3 processing the task

# Submitting Tasks

```
ExecutorService es2 = Executors.newSingleThreadExecutor();  
for (int i = 0; i < 6; i++) {  
    es2.submit(() -> System.out.println(Thread.currentThread().getName()  
        + " processing the task"));  
}  
es2.shutdown();
```

pool-2-thread-1 processing the task  
pool-2-thread-1 processing the task  
pool-2-thread-1 processing the task  
pool-2-thread-1 processing the task  
pool-2-thread-1 processing the task  
pool-2-thread-1 processing the task

# Submitting Tasks

```
ExecutorService es3 = Executors.newCachedThreadPool();  
for (int i = 0; i < 6; i++) {  
    es3.submit(() -> System.out.println(Thread.currentThread().getName()  
        + " processing the task"));  
}  
es3.shutdown();
```

pool-3-thread-2 processing the task  
pool-3-thread-3 processing the task  
pool-3-thread-1 processing the task  
pool-3-thread-4 processing the task  
pool-3-thread-5 processing the task  
pool-3-thread-6 processing the task



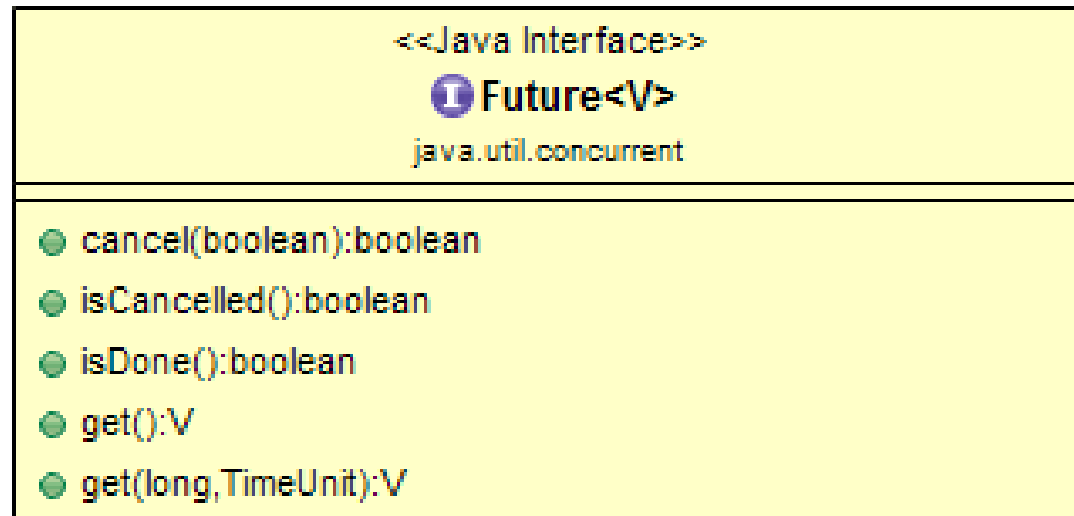
# Future

- A Future holds the result of an asynchronous computation.
- We get a Future object after submitting a runnable or a callable
- The owner of the Future object can obtain the result when it is ready.

```
Future<T> submit(Callable<T> task)  
Future<?> submit(Runnable task)
```

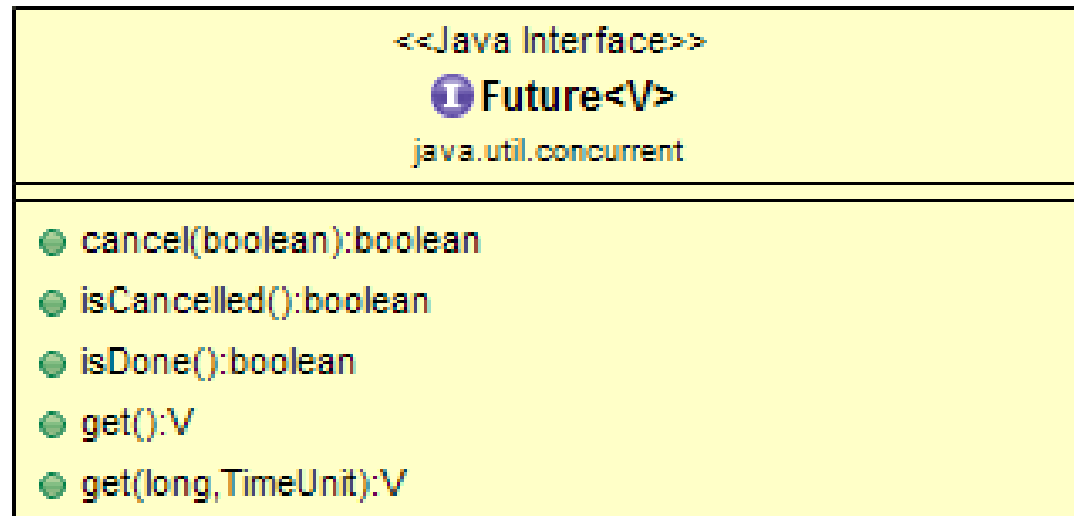
# Future

- `V get()`: Waits (blocks) if necessary for the computation to complete, and then retrieves its result.
- `V get(long timeout, TimeUnit unit)`: Waits if necessary (blocks) for at most the given time for the computation to complete, and then retrieves its result. Otherwise throws a `TimeoutException`



# Future

- `boolean cancel(boolean mayInterrupt)` - Attempts to cancel execution of this task.
- `boolean isCancelled()` - Returns true if this task was canceled before it completed normally.
- `boolean isDone()` - Returns true if this task completed.



# Using a Thread Pool

```
ExecutorService executor =  
Executors.newFixedThreadPool(4);  
  
// Define the task  
Callable<String> task = .....;  
  
// Submit the task  
Future<String> future = executor.submit(task);  
  
// Use Future to get the result asynchronously  
String result = future.get(); // may block  
  
executor.shutdown();
```

1. Call the static factory method of the Executors class to get a thread pool.
2. Define the task using Callable or Runnable
3. Call submit to submit Callable or Runnable objects.
4. Hang on to the returned Future objects so that you can get the results or cancel the tasks.
5. Call shutdown when you no longer want to submit any tasks.

# Using a Thread Pool

```
ExecutorService executor =  
Executors.newFixedThreadPool(4);
```

```
// Define the task  
Runnable<String> task = .....;
```

```
// Execute the task  
executor.execute(task);
```

```
executor.shutdown();
```

`execute(Runnable)` is like `submit(Runnable)` with the following difference:

- `execute` not return anything (returns **void**)
- `submit` returns a **Future** object to manage the task (e.g., cancel).

# Controlling Groups of Tasks

- We have seen how to use an executor service as a thread pool to increase the efficiency of task execution
- We can also use an executor to control a group of related tasks

```
T                invokeAny(Collection<? extends Callable<T>> tasks)
```

```
List<Future<T>>  invokeAll(Collection<? extends Callable<T>> tasks)
```

# Controlling Groups of Tasks

- The `invokeAny` method submits all objects in a **collection** of `Callable` objects and returns **the result of a completed task**.
- You don't know which task that is—presumably, it is the one that finished most quickly.
- Use this method for a search problem in which you are willing to accept any solution (e.g., many threads break the same cipher).

```
ExecutorService executorService =  
    Executors.newFixedThreadPool(3);  
  
Set<Callable<String>> callables = new HashSet<>();  
  
callables.add(() -> "Task 1");  
callables.add(() -> "Task 2");  
callables.add(() -> "Task 3");  
  
String result = executorService.invokeAny(callables);  
  
System.out.println("result = " + result);  
  
executorService.shutdown();
```

# Controlling Groups of Tasks

- The `invokeAll` method submits all objects in a **collection** of Callable objects,
- This method blocks until all of the tasks are complete (i.e., `Future.isDone()` is true for each element of the returned list)
- The method returns a list of Future objects that represent the solutions to all tasks.
- Tasks might terminate normally or exceptionally, Futures can wrap the exception.

```
ExecutorService executorService = Executors.newFixedThreadPool(2);

List<Callable<String>> callables = new ArrayList<>();

callables.add(() -> "Task 1");
callables.add(() -> "Task 2");
callables.add(() -> "Task 3");
callables.add(() -> "Task 4");

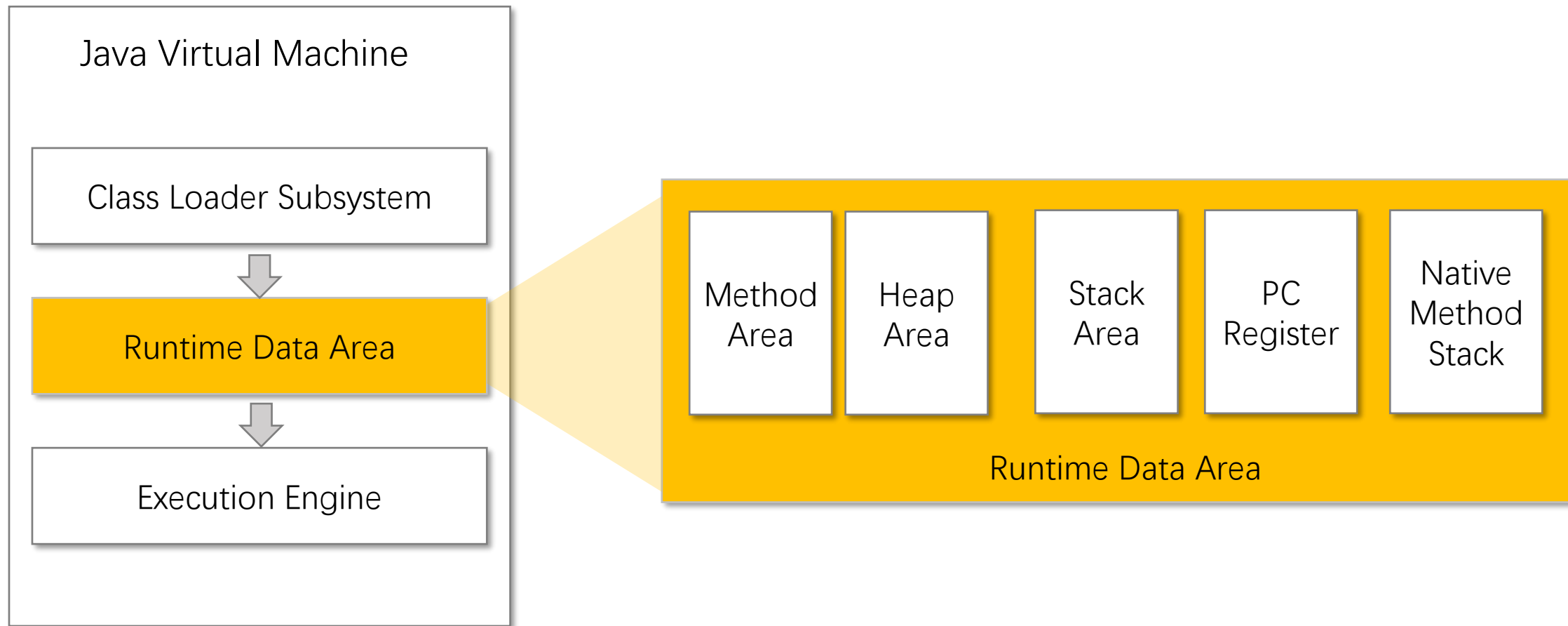
List<Future<String>> futures = executorService.invokeAll(callables);

for(Future<String> future : futures){
    System.out.println("future.get = " + future.get());
}

executorService.shutdown();
```



# JVM & Threading



# Shared between Threads



- Method Area
  - stores class and interface definitions
  - created on JVM start-up
  - shared among all JVM threads
- Heap
  - where all class instances and arrays (`new XXX()`) are allocated
  - created on JVM start-up
  - shared among all JVM threads

<https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html>

<https://blog.jamesdbloom.com/JVMInternals.html>

# Per Thread

- **PC registers**

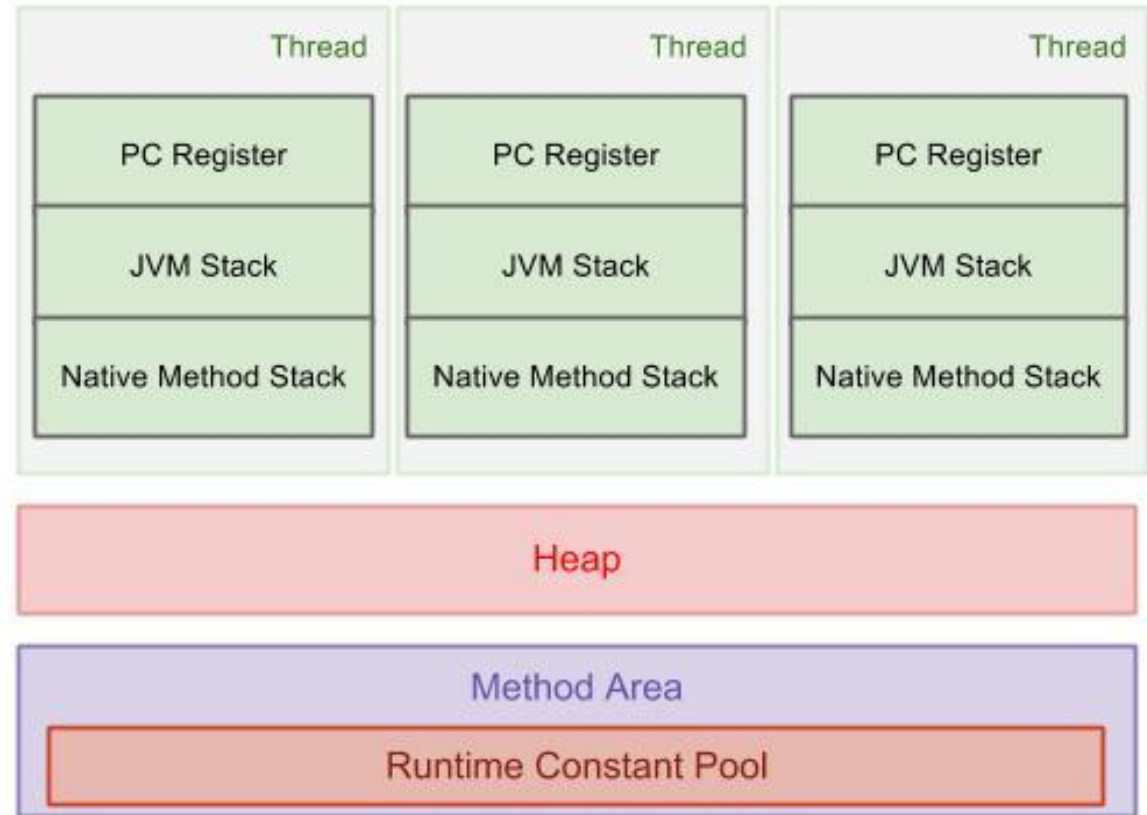
- contains the address of the Java Virtual Machine instruction currently being executed

- **Stacks**

- Each thread has its own stack, created at the same time as the thread
- holds a frame for each method and contains local variables and partial results

- **Native Method Stacks**

- support native methods (methods written in a language other than Java, such as C/C++).
- allocated per thread when each thread is created



<https://www.programcreek.com/wp-content/uploads/2013/04/JVM-runtime-data-area.jpg>

# Next Lecture

- Network Programming