

# Computer System Design & Application

## 计算机系统设计与应用A

陶伊达 (TAO Yida)

taoyd@sustech.edu.cn



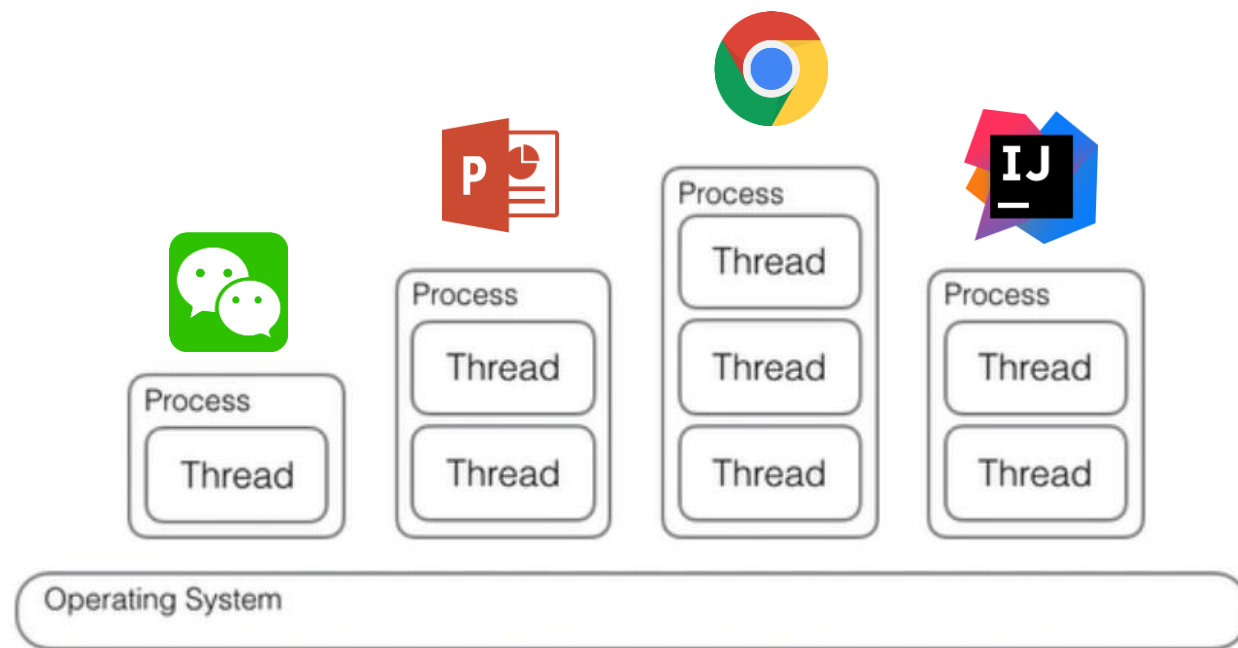
# Lecture 8

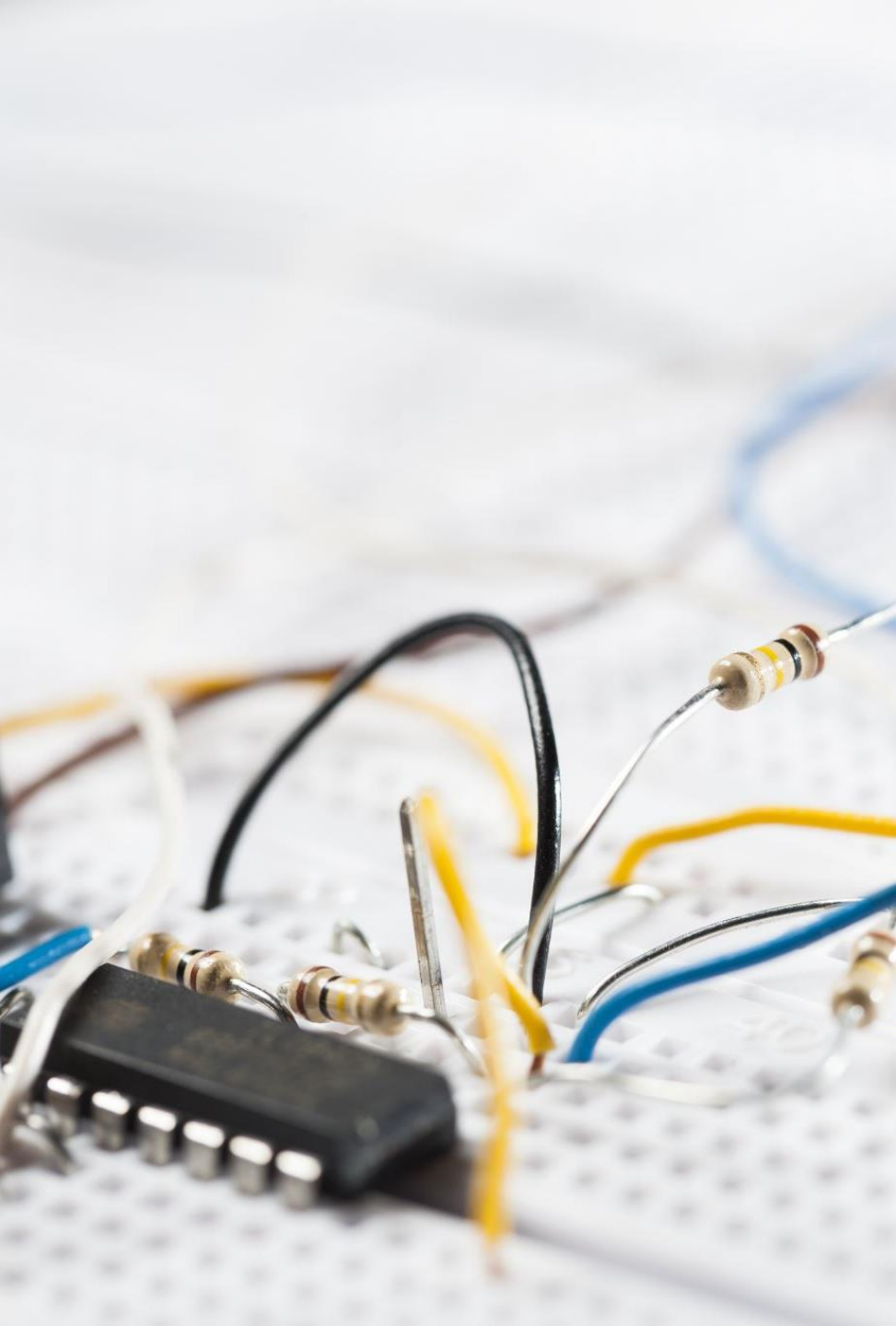
---

- Multithreading Overview
- Creating & Starting Threads
- Thread Safety
- Concurrent Collections

# 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.



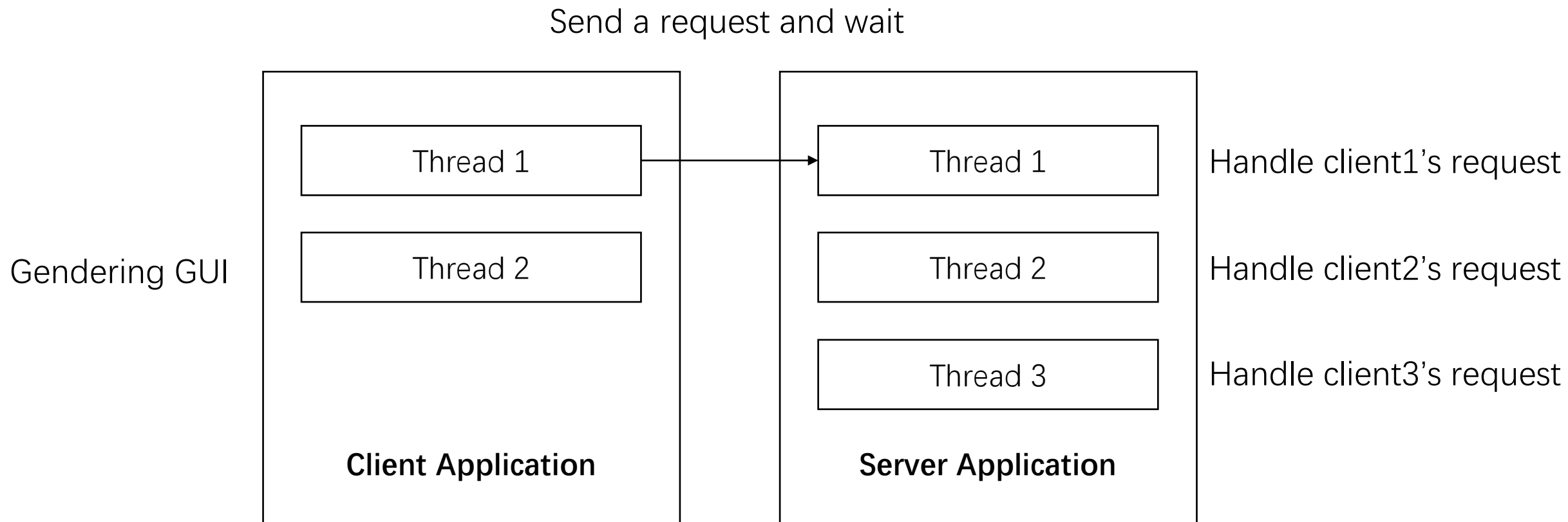


# Multithreading

---

- In Java, Multithreading refers to a process of executing two or more threads simultaneously for maximum utilization of the CPU.
- 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.

# Multithreading



# Multithreading in Java

- The main thread is created automatically when our Java program is started

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

Output "main"



An abstract graphic on the left side of the slide, featuring concentric circles and various digital patterns like binary code and pixelated shapes in shades of blue, green, and white.

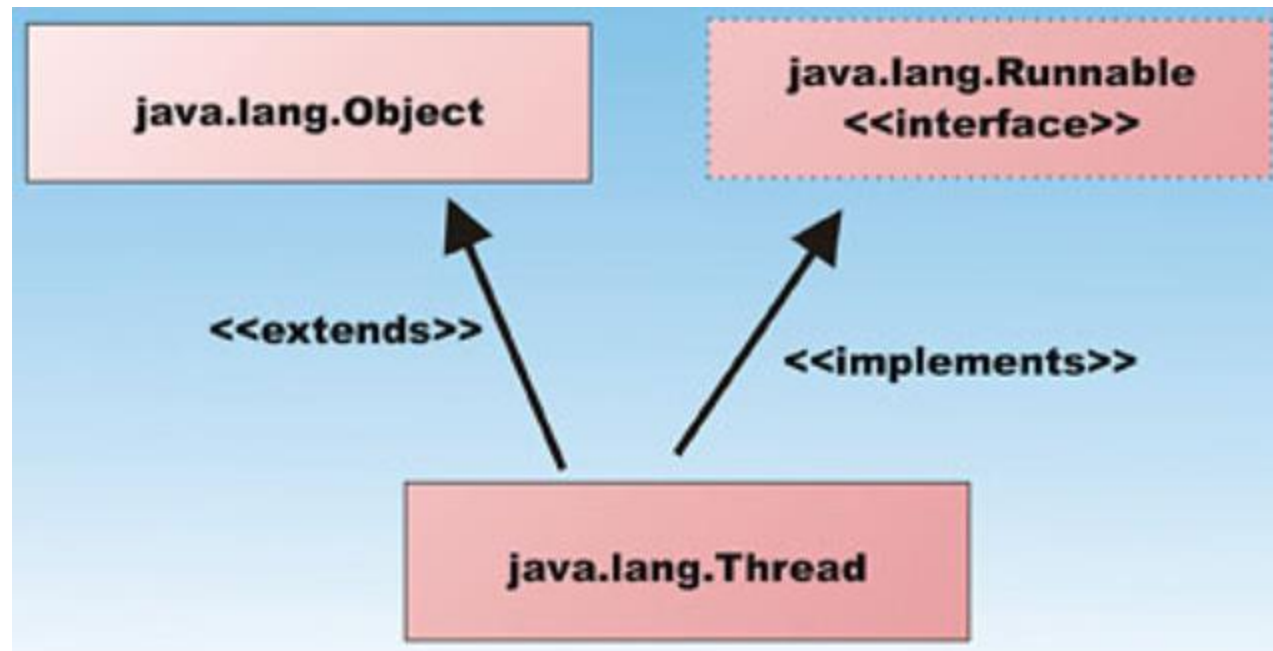
# Lecture 8

---

- Multithreading Overview
- Creating & Starting Threads
- Thread Safety
- Concurrent Collections

# Creating & Starting Threads

- Approach 1: Extending the `Thread` class
- Approach 2: Implementing the `Runnable` interface (preferred)





# The Thread Class

```
public class Thread
    extends Object
    implements Runnable
```

- One way to create a new thread of execution is to declare a class to be a subclass of `Thread`
- This subclass should override the `run` method of class `Thread`; specify what this thread does inside `run`.
- An instance of the subclass can then be allocated and started

```
public class CatThread extends Thread{
    @Override
    public void run() {
        System.out.println("I'm a cat.");
    }
}
```

```
public class Concurrency {
    public static void main(String[] args){
        Thread cat = new CatThread();
        cat.start();
    }
}
```

1. How many threads? 2. why start()?

# Using Thread

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

    @Override
    public void run() {
        while (cnt<10){
            System.out.println(Thread.currentThread().getName()
                               + ": I'm a cat " + (++cnt));
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

- Print a string 10 times
- 1s interval between each print
- Also print the current thread's name at the same time

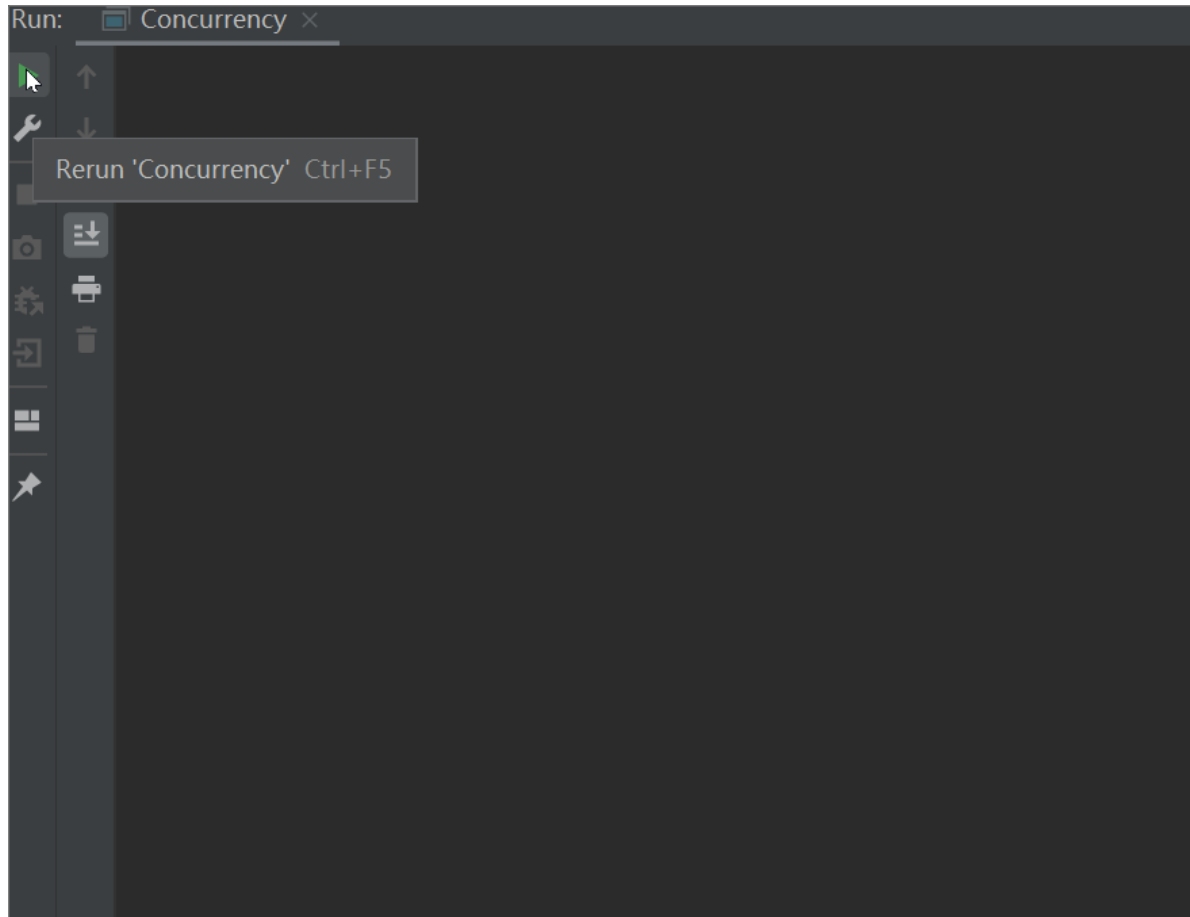
# Using Thread (cont.)

```
public class Concurrency {  
    public static void main(String[] args) throws InterruptedException {  
        Thread cat = new CatThread();  
        cat.start();  
  
        int cnt = 0;  
        while(cnt<10){  
            System.out.println(Thread.currentThread().getName() + (++cnt));  
            Thread.sleep(1000);  
        }  
    }  
}
```

What will happen?

- Print the current thread's name for 10 times
- 1s interval between each print

# Using Thread (cont.)



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

# Using Thread (cont.)

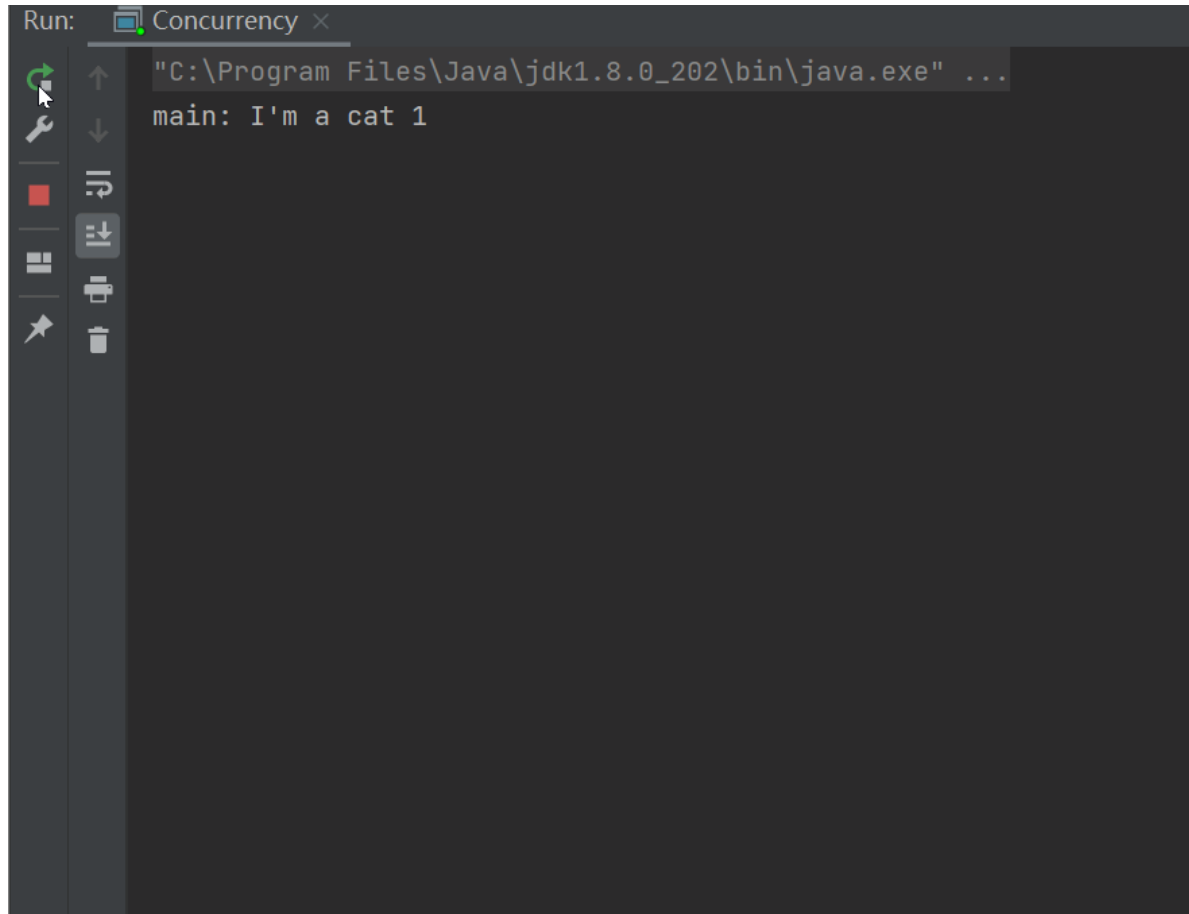
Why start()?

```
public class Concurrency {  
    public static void main(String[] args) throws InterruptedException {  
        Thread cat = new CatThread();  
        cat.start();  
  
        int cnt = 0;  
        while(cnt < 10){  
            System.out.println(Thread.currentThread().getName() + (++cnt));  
            Thread.sleep(1000);  
        }  
    }  
}
```

**Non-blocking!**  
Don't have to wait for it before  
executing the subsequent operations



# Why start() instead of run()?



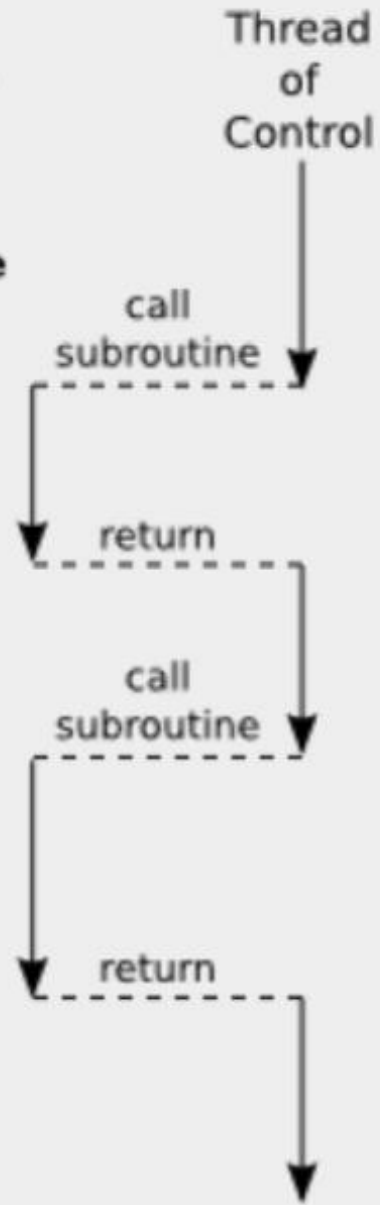
```
Thread cat = new CatThread();  
//cat.start();  
cat.run();
```

## Observation

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

- **run() executes like a normal method**
- **start() executes certain native code to start a new thread**

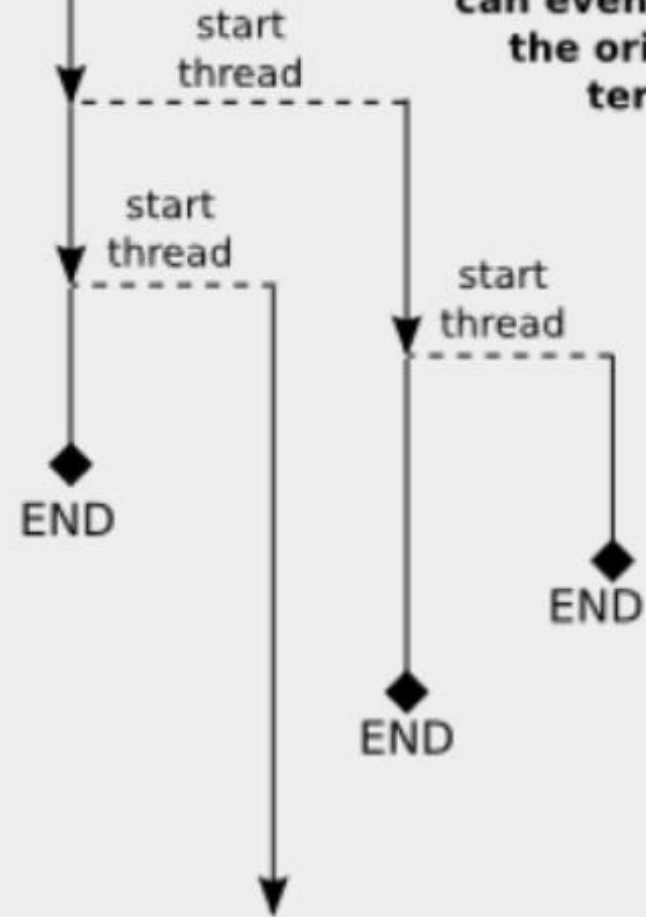
When a thread calls a subroutine, there is still only one thread of control, which is in the subroutine for a time until the subroutine returns.



Time



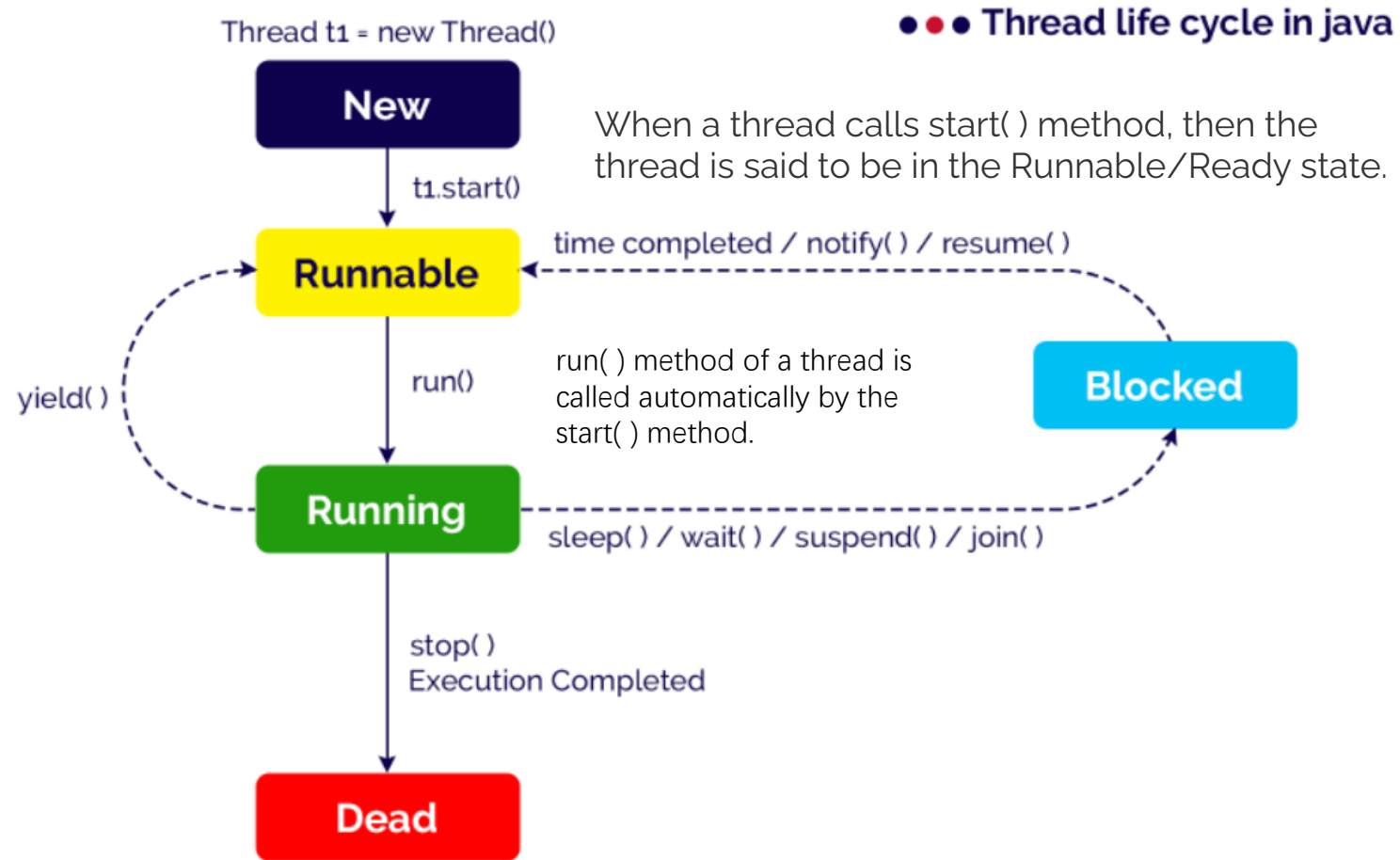
Thread of Control



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

# Thread States & Lifecycle



# 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()

```
@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();
}
```

# Implementing Runnable

## 1. Using Class

```
public class RunnableThread implements Runnable{  
    public void run(){  
        System.out.println("This is a runnable.");  
    }  
}  
  
Runnable runnable1 = new RunnableThread();
```

## 2. Using Anonymous Class

```
Runnable runnable2 = new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("This is a runnable.");  
    }  
};
```

## 3. Using Lambda Expressions

```
Runnable runnable3 = () -> System.out.println("This is a runnable");
```



# Starting a Thread with a Runnable

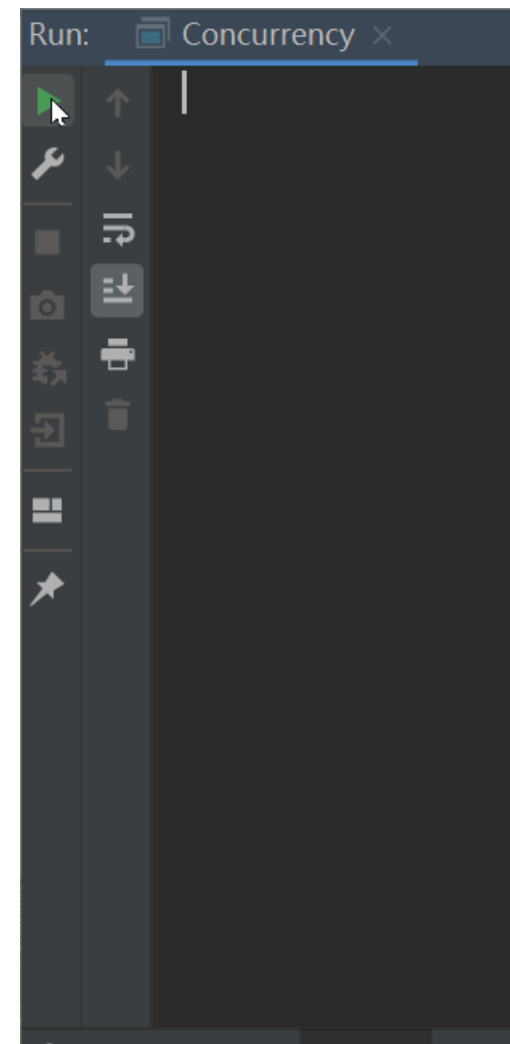
- Thread has a constructor that takes a Runnable `Thread(Runnable target)`  
Allocates a new Thread object.
- To have the `run()` method executed by a thread, pass an instance of a class, anonymous class or lambda expression that implements the Runnable interface to a Thread constructor

```
Runnable runnable = () -> System.out.println(Thread.currentThread().getName());

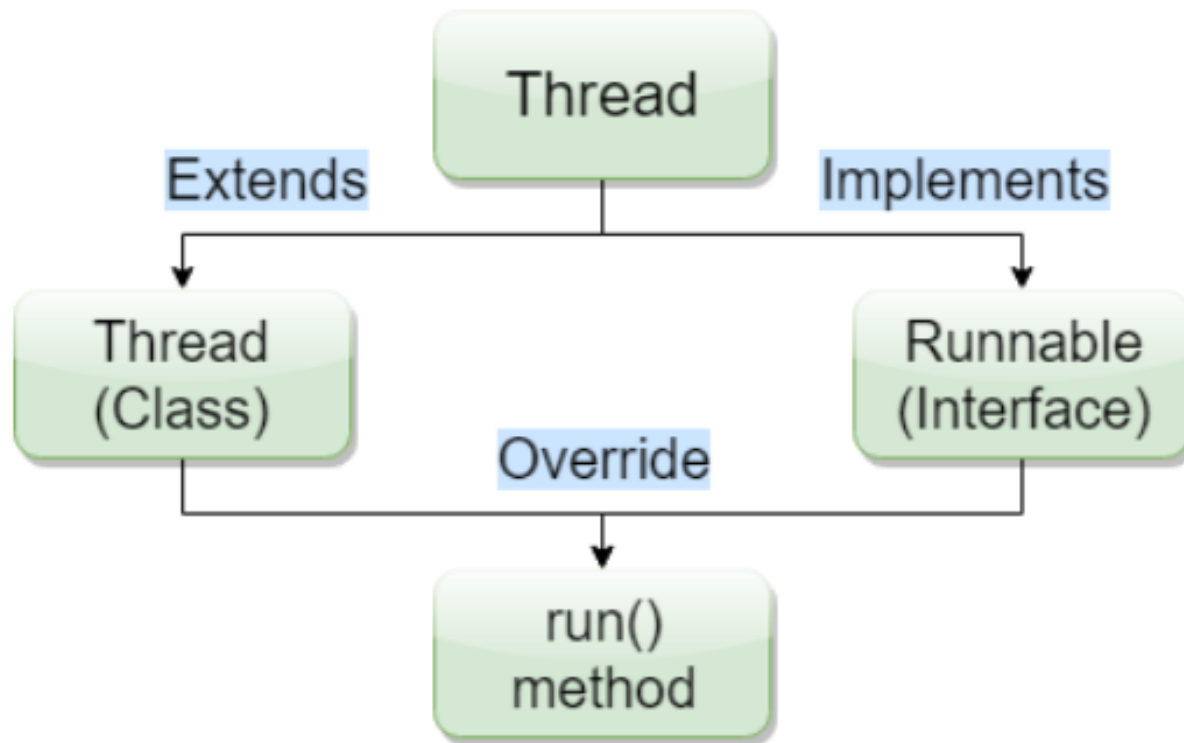
Thread runnableThread = new Thread(runnable);
runnableThread.start();
```

# 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 = new DogThread();  
    Thread dog = new Thread(runnable);  
    dog.start();  
  
    // Main  
    int cnt = 0;  
    while(cnt<10){  
        System.out.println(Thread.currentThread().getName() + (++cnt));  
        Thread.sleep(1000);  
    }  
}
```



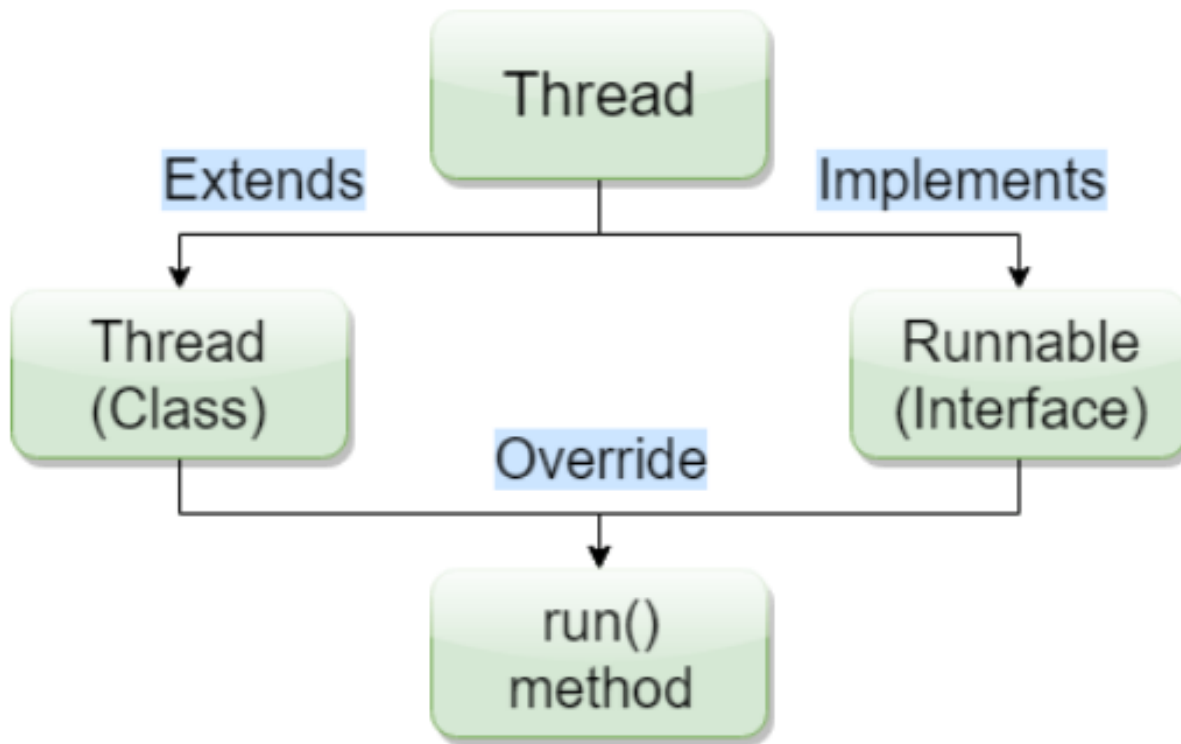
# Subclass vs Runnable



## 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` separates the task from the `Thread` object that executes the task

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 8

---

- Multithreading Overview
- Creating & Starting Threads
- Thread Safety
- Concurrent Collections



# Example: shared resource

```
public class DogThread implements Runnable{
    private int bones = 10;

    public void run(){
        while (bones > 0 ){
            System.out.println(Thread.currentThread().getName()
                               + ": Dog eats bone " + (bones--));
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

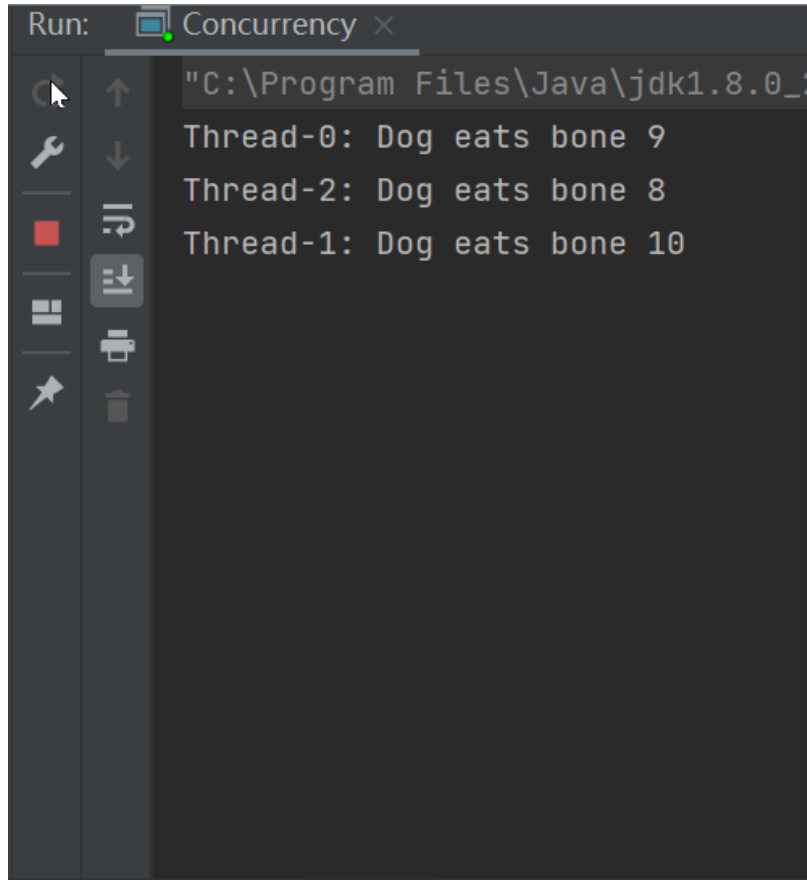
## Inside main()

```
Runnable dog = new DogThread();
new Thread(dog).start();
new Thread(dog).start();
new Thread(dog).start();
```

## What if we extend Thread?

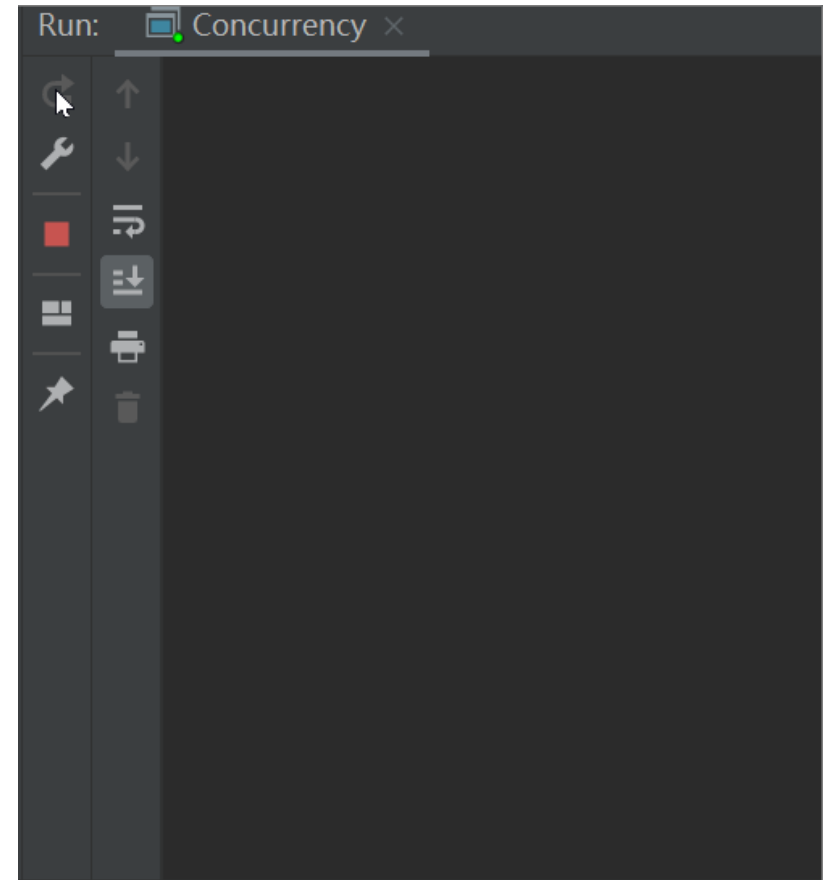
```
Thread dog1 = new DogThread();
Thread dog2 = new DogThread();
Thread dog3 = new DogThread();
```

# Example: shared resource



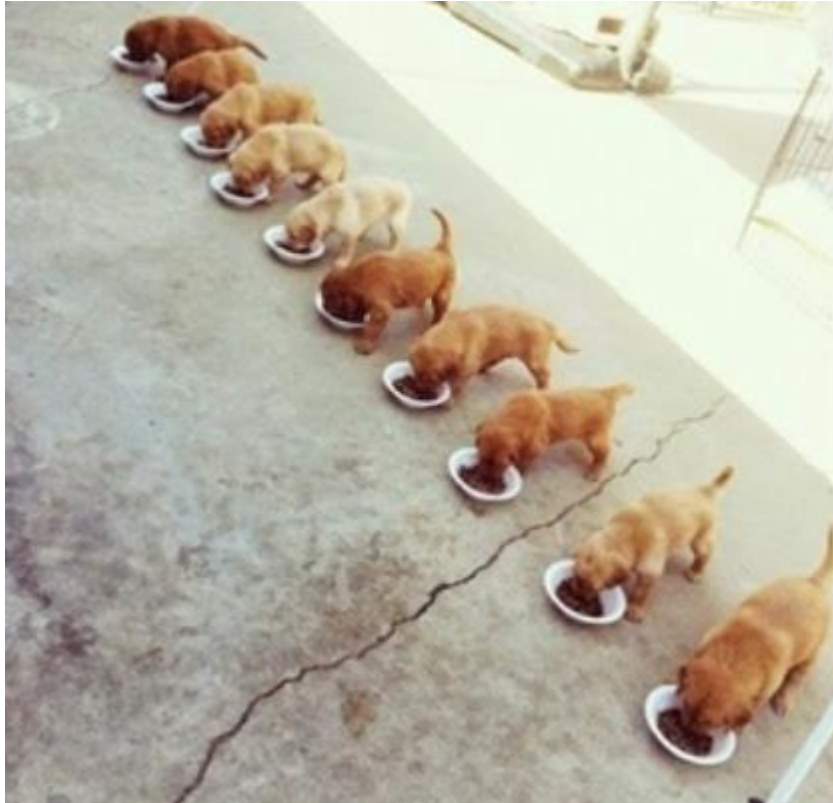
```
Run: Concurrency ×  
"C:\Program Files\Java\jdk1.8.0_2  
Thread-0: Dog eats bone 9  
Thread-2: Dog eats bone 8  
Thread-1: Dog eats bone 10
```

The same bone has been eaten by multiple dogs



-1 bone?

# Example: shared resource



What we want



What we get

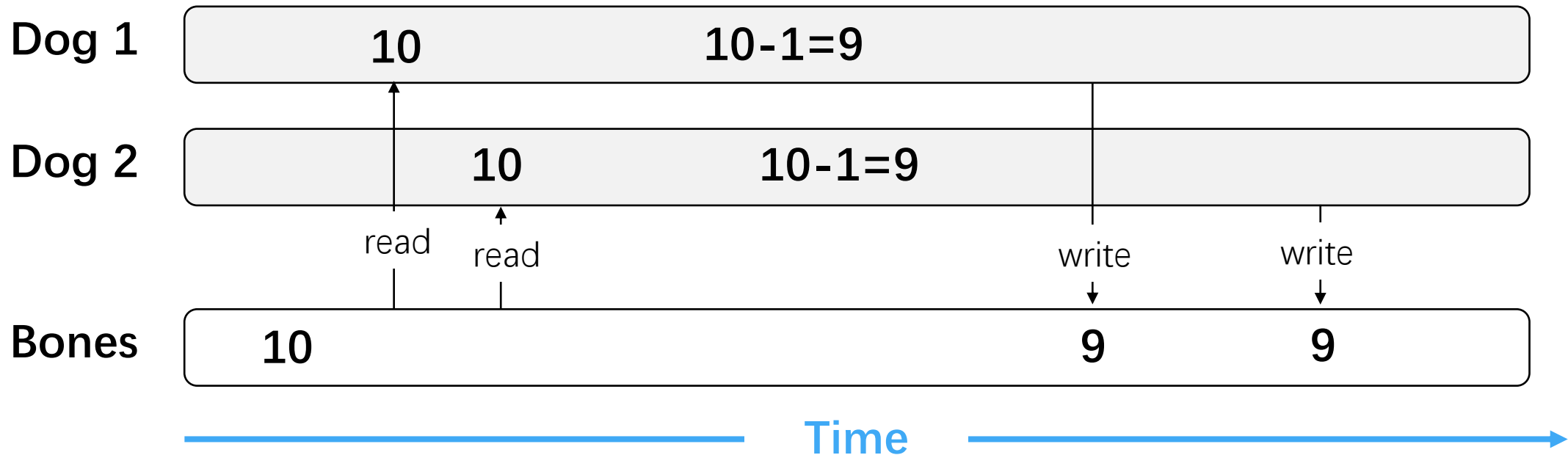


# Race Condition

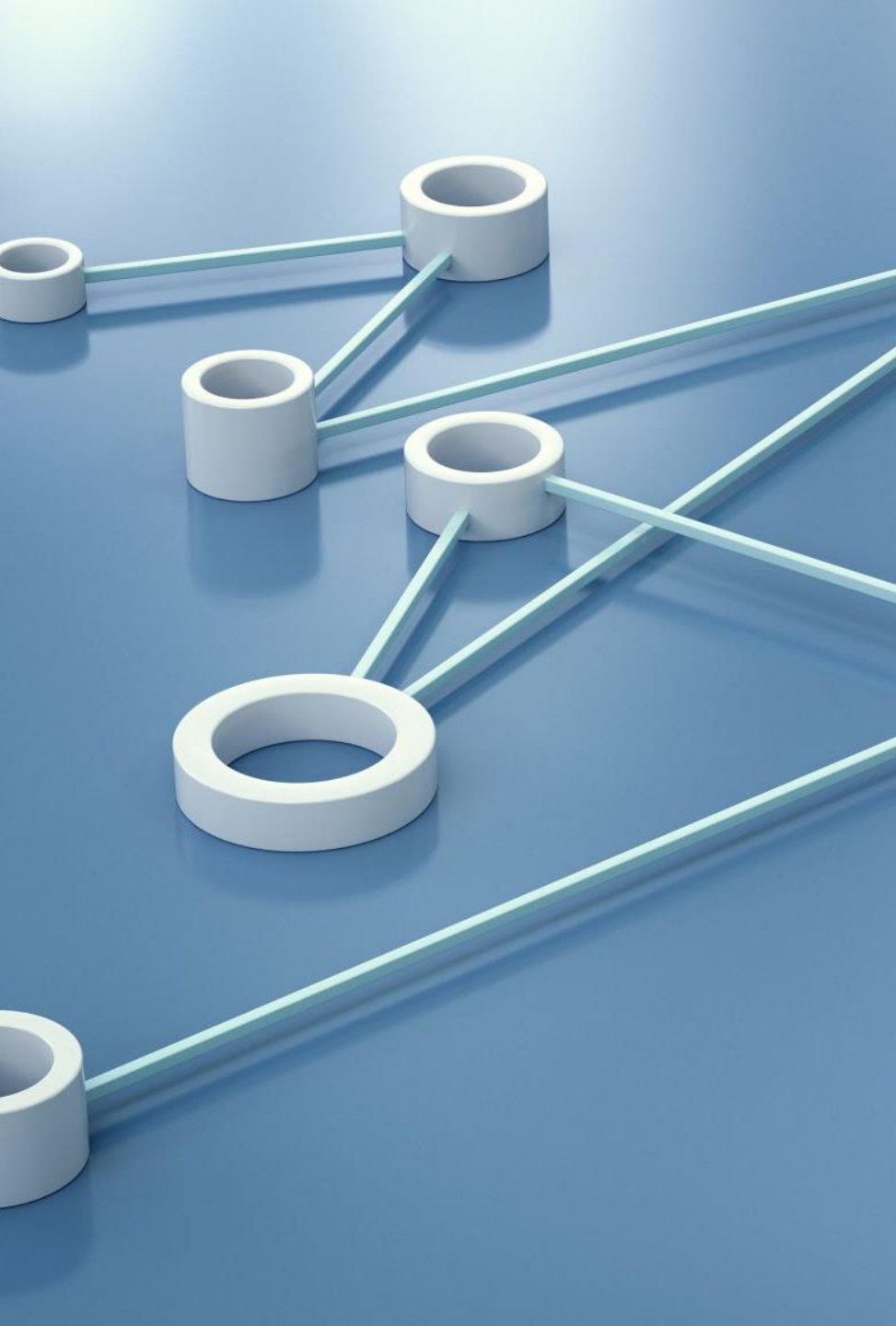
- A concurrency problem/bug
- Multiple threads compete for a shared resource (race)
- The final results depend on which thread gets the resource first (non-deterministic)

# 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 `synchronized` keyword
  - The `Concurrency` API (`java.util.concurrent`), introduced in Java 5

# Using the synchronized Keyword

- A synchronized block is wrapped using the synchronized keyword
  - A code block inside a method (同步代码块)
  - A method (同步方法)
- All synchronized blocks synchronized on the same object can only have one thread executing inside them at the same time.
- All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

# Synchronized Methods

```
public class DogThread implements Runnable{
    private int bones = 10;
    private boolean hasBone = true;

    public synchronized void eat(){
        if(bones <= 0){
            hasBone = false;
        } else{
            System.out.println(Thread.currentThread().getName()
                               + ": Dog eats bone " + (bones--));
        }
    }

    public void run(){
        while (hasBone){
            eat();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
Runnable dog = new DogThread();
new Thread(dog).start();
new Thread(dog).start();
new Thread(dog).start();
```

Run: Concurrency ×

"C:\Program Files\Java\jdk1

Thread-0: Dog eats bone 10  
Thread-2: Dog eats bone 9  
Thread-1: Dog eats bone 8

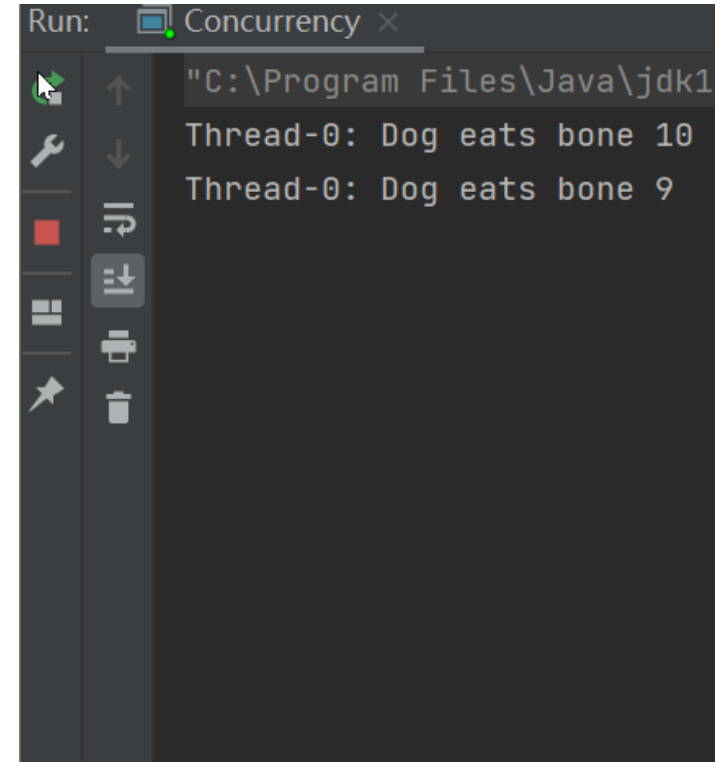
# Can we synchronize the run() method?

```
public class DogThread implements Runnable{
    private int bones = 10;
    private boolean hasBone = true;

    public void eat(){
        if(bones <= 0){
            hasBone = false;
        } else{
            System.out.println(Thread.currentThread().getName()
                               + ": Dog eats bone " + (bones--));
        }
    }

    public synchronized void run(){
        while (hasBone){
            eat();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

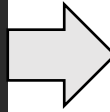
```
Runnable dog = new DogThread();
new Thread(dog).start();
new Thread(dog).start();
new Thread(dog).start();
```



```
Run: Concurrency x
"C:\Program Files\Java\jdk1
Thread-0: Dog eats bone 10
Thread-0: Dog eats bone 9
```

# Synchronized Code Block

```
public synchronized void eat(){
    if(bones <= 0){
        hasBone = false;
    } else{
        System.out.println(Thread.currentThread().getName()
            + ": Dog eats bone " + (bones--));
    }
}
```



```
public void eat(){
    synchronized(this) {
        if (bones <= 0) {
            hasBone = false;
        } else {
            System.out.println(Thread.currentThread().getName()
                + ": Dog eats bone " + (bones--));
        }
    }
}
```

# Using Lock in the Concurrency API

- Java 5 added a new Java package `java.util.concurrent`, which contains a set of classes that makes it easier to develop concurrent (multithreaded) applications in Java
- The Java Lock interface, `java.util.concurrent.locks.Lock`, represents a concurrent lock which can be used to guard against race conditions inside critical sections.
- The Lock interface is more flexible and provides more options than a synchronized block

# A more practical problem: Bank Account Management

```
BankAccount
  BankAccount()
  deposit(double): void
  withdraw(double): void
  balance: double
```

```
DepositRunnable
  Runnable
  run(): void
  DepositRunnable(BankAccount, double, int)
  DELAY: int = 1
  account: BankAccount
  amount: double
  count: int
```

```
public void run() {
    try {
        for (int i = 1; i <= count; i++) {
            account.deposit( amount );
            Thread.sleep( DELAY );
        }
    } catch (InterruptedException exception) {}
}
```

```
WithdrawRunnable
  Runnable
  run(): void
  WithdrawRunnable(BankAccount, double, int)
  DELAY: int = 1
  account: BankAccount
  amount: double
  count: int
```

```
public void run() {
    try {
        for (int i = 1; i <= count; i++) {
            account.withdraw( amount );
            Thread.sleep( DELAY );
        }
    } catch (InterruptedException exception) {}
}
```



# 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

```
Lock lock = new ReentrantLock();
```

# Using Lock

- To lock the Lock instance, invoke its lock() method
- To unlock the Lock instance, invoke its unlock() method

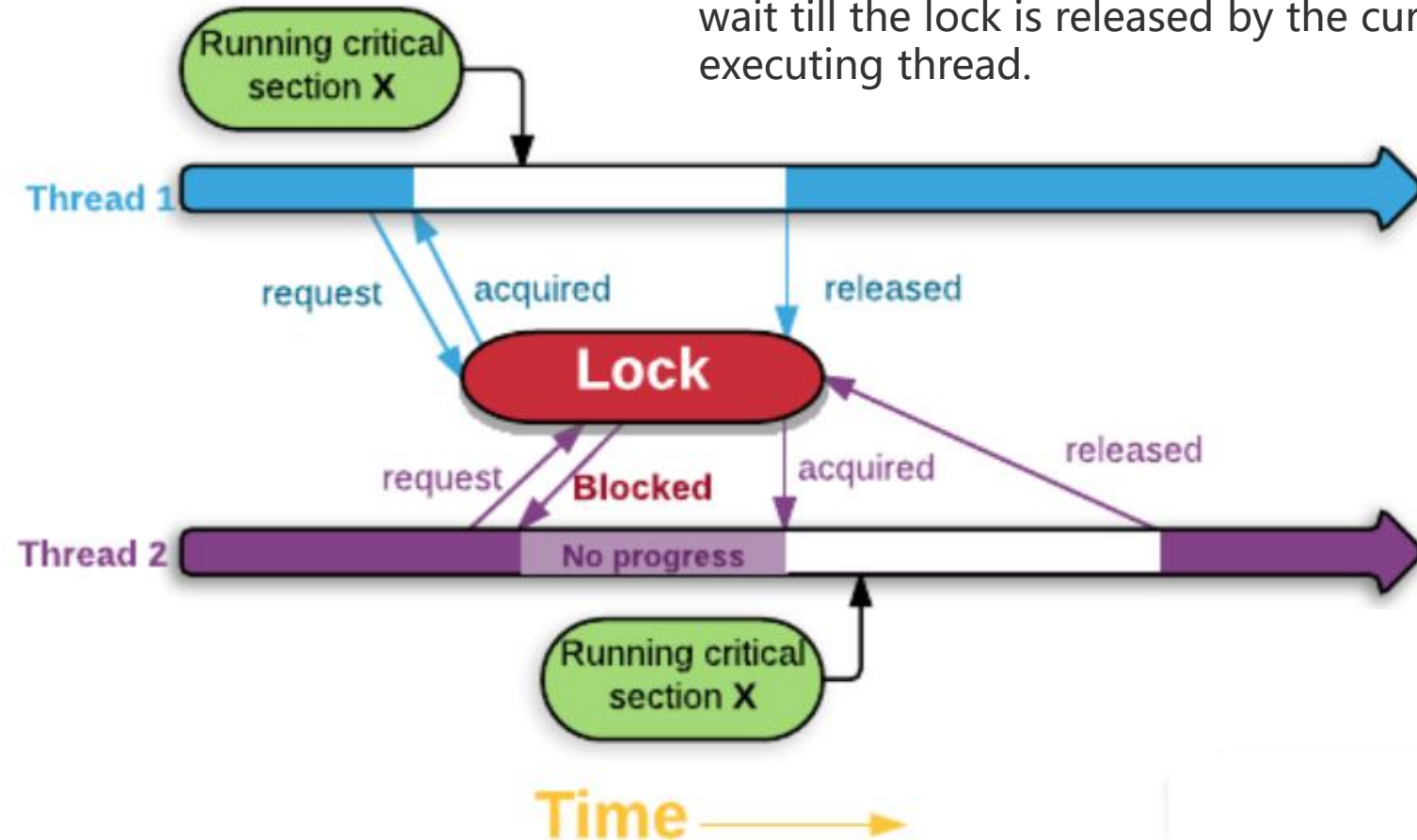
```
public class BankAccount {  
    private Lock balanceChangeLock;  
    ...  
    public BankAccount() {  
        balanceChangeLock = new ReentrantLock();  
        ...  
    }  
}
```

```
balanceChangeLock.lock();  
Manipulate the shared resource.  
balanceChangeLock.unlock();
```

- 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.

## Mutual Exclusion of Critical Section

- As long as a thread owns a lock, no other thread can acquire the same lock.
- The other threads will block when they attempt to acquire the lock. The blocked threads will wait till the lock is released by the currently executing thread.



<https://www.logicbig.com/tutorials/core-java-tutorial/java-multi-threading/java-intrinsic-locks.html>

# Potential Flaw?

---

```
balanceChangeLock.lock();  
Manipulate the shared resource.  
balanceChangeLock.unlock();
```

- What will happen if the code between lock() and unlock() throws an exception?
  - The call unlock() never happen
  - The current thread continues to hold the lock, and no other thread can acquire it

# Avoid Exceptions lock a Lock forever

- To overcome this problem, place unlock() in a finally clause

```
public void deposit (double amount) {  
    balanceChangeLock.lock();  
    try {  
        System.out.print("Depositing " + amount);  
        double newBalance = balance + amount;  
        System.out.println(", new balance is " + newBalance);  
        balance = newBalance;  
    } finally {  
        balanceChangeLock.unlock();  
    }  
}
```

- The finally block *always* executes when the try block exits.
- This ensures that the finally block is executed even if an unexpected exception occurs

# Deadlock

- Thread A acquires a lock and then waits for thread B to do some essential work.
- Thread B is currently waiting to acquire the same lock in order to do the essential work

Interviewer: “Explain deadlock to us and we’ll hire you.”

Me: “Hire me and I’ll explain it to you.”

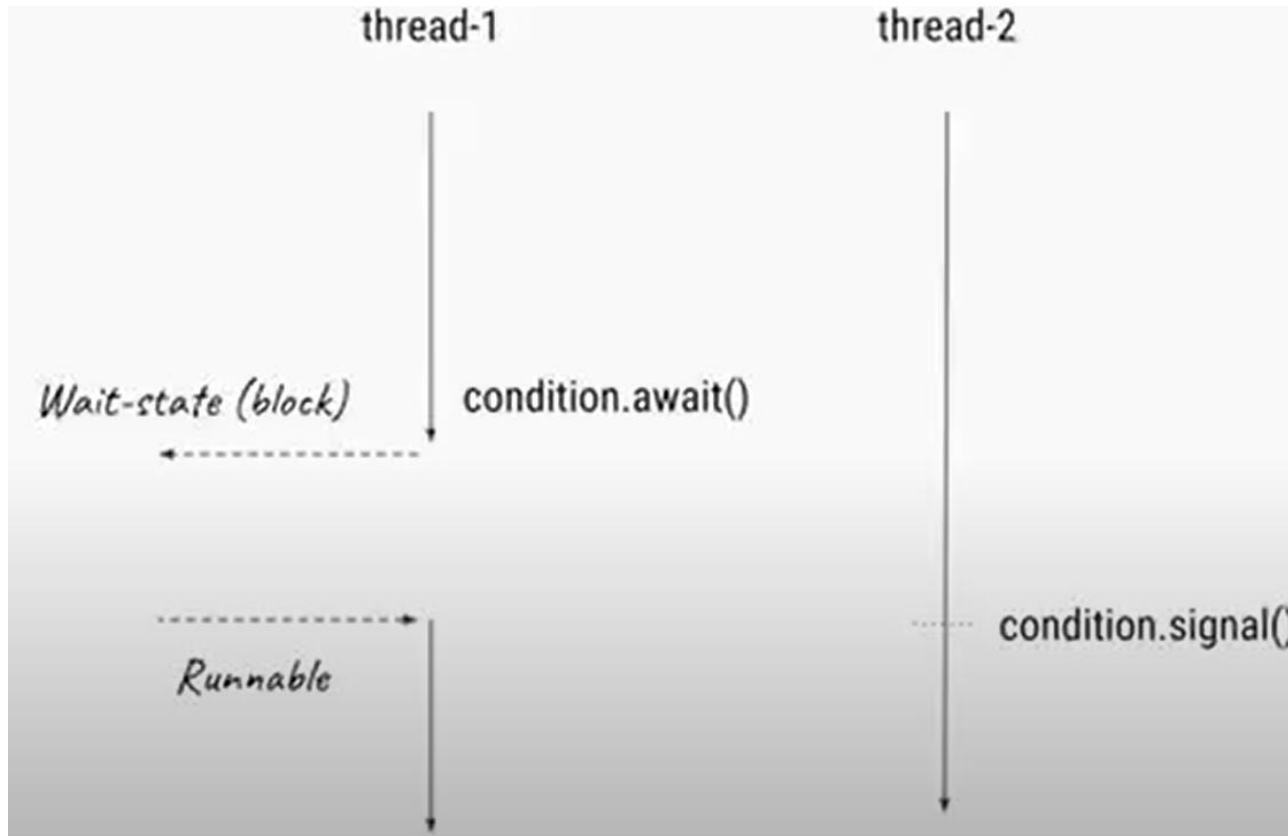
# Deadlock

- To disallow negative balance during withdraw, we can wait for other threads to deposit money
- Can we use `sleep()` to wait?
  - 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.
  - We could obtain a condition object with the `newCondition()` method of the `Lock` interface
- 
- A Condition object is necessarily bound to a Lock
  - It is customary to give the condition object a name that describes the condition that you want to test

```
public class BankAccount {  
    private Lock balanceChangeLock;  
    private Condition sufficientFundsCondition;  
    . . .  
    public BankAccount() {  
        balanceChangeLock = new ReentrantLock();  
        sufficientFundsCondition = balanceChangeLock.newCondition();  
        . . .  
    }  
}
```

# Using Condition await()

When calling await, this thread is in a blocked state, and it will not be activated by the thread scheduler until it is unblocked.

- For a condition to take effect, we need to implement an appropriate test (i.e., condition)
- For as long as the test/condition is not fulfilled, call the await() method on the condition object
- Calling await() on a condition object makes the current thread wait and allows another thread to acquire the lock object.

```
public void withdraw(double amount)
{
    balanceChangeLock.lock();
    try
    {
        while (balance < amount)
        {
            sufficientFundsCondition.await();
        }
        . . .
    }
    finally
    {
        balanceChangeLock.unlock();
    }
}
```

# Using Condition `signalAll()`



The call to `signalAll()` notifies the waiting threads that sufficient funds may be available, and that it is worth testing the loop condition again



- 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
    {
        . . .
        sufficientFundsCondition.signalAll();
    }
    finally
    {
        balanceChangeLock.unlock();
    }
}
```



# To Put it Altogether



```
public class BankAccount {  
    private double balance;  
    private Lock balanceChangeLock;  
    private Condition sufficientFundsCondition;  
  
    /**  
     * Constructs a bank account with a zero balance.  
     */  
    public BankAccount() {  
        balance = 0;  
        balanceChangeLock = new ReentrantLock();  
        sufficientFundsCondition = balanceChangeLock.newCondition();  
    }  
}
```



  BankAccount



  BankAccount()

  deposit(double): void

  withdraw(double): void

  balance: double

  balanceChangeLock: Lock

  sufficientFundsCondition: Condition

# To Put it Altogether

```
public void withdraw (double amount) throws InterruptedException {
    balanceChangeLock.lock();
    try {
        while (balance < amount) {
            sufficientFundsCondition.await();
        }
        System.out.print( "Withdrawing " + amount );
        double newBalance = balance - amount;
        System.out.println( ", new balance is " + newBalance );
        balance = newBalance;
    } finally {
        balanceChangeLock.unlock();
    }
}
```

```
public void deposit (double amount) {
    balanceChangeLock.lock();
    try {
        System.out.print( "Depositing " + amount );
        double newBalance = balance + amount;
        System.out.println( ", new balance is " + newBalance );
        balance = newBalance;
        sufficientFundsCondition.signalAll();
    } finally {
        balanceChangeLock.unlock();
    }
}
```

# To Put it Altogether

```
DepositRunnable
  Runnable
    run(): void
  DepositRunnable(BankAccount, double, int)
  DELAY: int = 1
  account: BankAccount
  amount: double
  count: int
```

```
public void run() {
    try {
        for (int i = 1; i <= count; i++) {
            account.deposit( amount );
            Thread.sleep( DELAY );
        }
    } catch (InterruptedException exception) {}
}
```

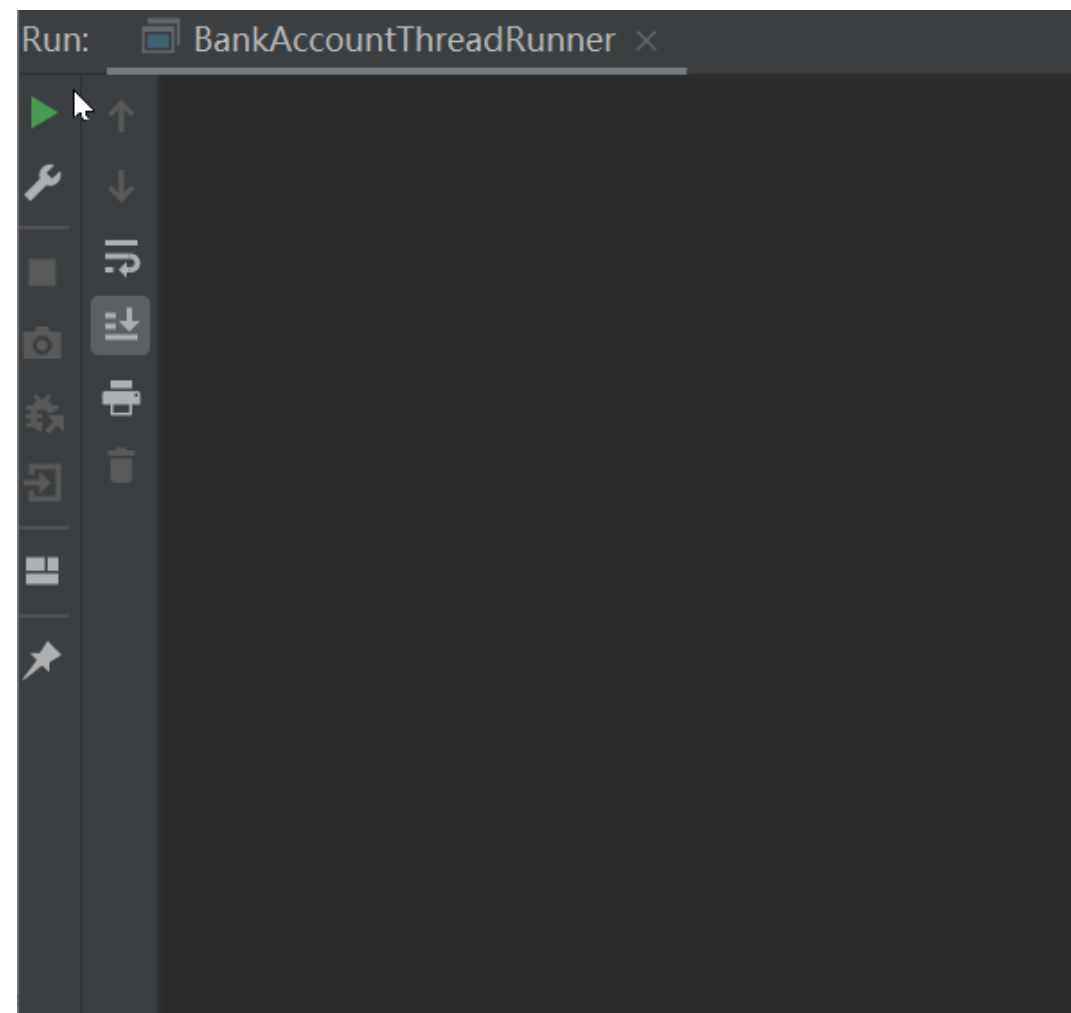
```
WithdrawRunnable
  Runnable
    run(): void
  WithdrawRunnable(BankAccount, double, int)
  DELAY: int = 1
  account: BankAccount
  amount: double
  count: int
```

```
public void run() {
    try {
        for (int i = 1; i <= count; i++) {
            account.withdraw( amount );
            Thread.sleep( DELAY );
        }
    } catch (InterruptedException exception) {}
}
```



# To Put it Altogether

```
public class BankAccountThreadRunner {  
    public static void main (String[] args) {  
        BankAccount account = new BankAccount();  
        final double AMOUNT = 100;  
        final int REPETITIONS = 100;  
        final int THREADS = 100;  
  
        for (int i = 1; i <= THREADS; i++) {  
            DepositRunnable d =  
                new DepositRunnable( account, AMOUNT, REPETITIONS );  
            WithdrawRunnable w =  
                new WithdrawRunnable( account, AMOUNT, REPETITIONS );  
  
            Thread dt = new Thread(d);  
            Thread wt = new Thread(w);  
  
            dt.start();  
            wt.start();  
        }  
    }  
}
```



# Lock vs synchronized block

Using the synchronized block	Using Lock
A synchronized block is fully contained within a method.	We can have Lock API's lock() and unlock() operation in separate methods
A synchronized block doesn't support the fairness (i.e., a waiting thread could risk waiting forever).	We can achieve fairness within the Lock APIs by specifying the fairness property (i.e., makes sure that longest waiting thread is given access to the lock)
A thread gets blocked if it can't get an access to the synchronized block.	The Lock API provides tryLock() method. The thread acquires lock only if it's available and not held by any other thread.
A thread which is in “waiting” state to acquire the access to synchronized block, can't be interrupted.	The Lock API provides a method lockInterruptibly() which can be used to interrupt the thread when it's waiting for the lock.



# Lecture 8

---

- Multithreading Overview
- Creating & Starting Threads
- Thread Safety
- **Concurrent Collections**

# 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>

# Example: Adding to ArrayList with Multithreading

```
public class ArrayListThread implements Runnable{
    private List<Integer> numList;

    public ArrayListThread(List<Integer> numList){
        this.numList = numList;
    }
    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            numList.add(i);
            System.out.println(Thread.currentThread().
                getName() + ": added " + i);
            try {
                Thread.sleep(5);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

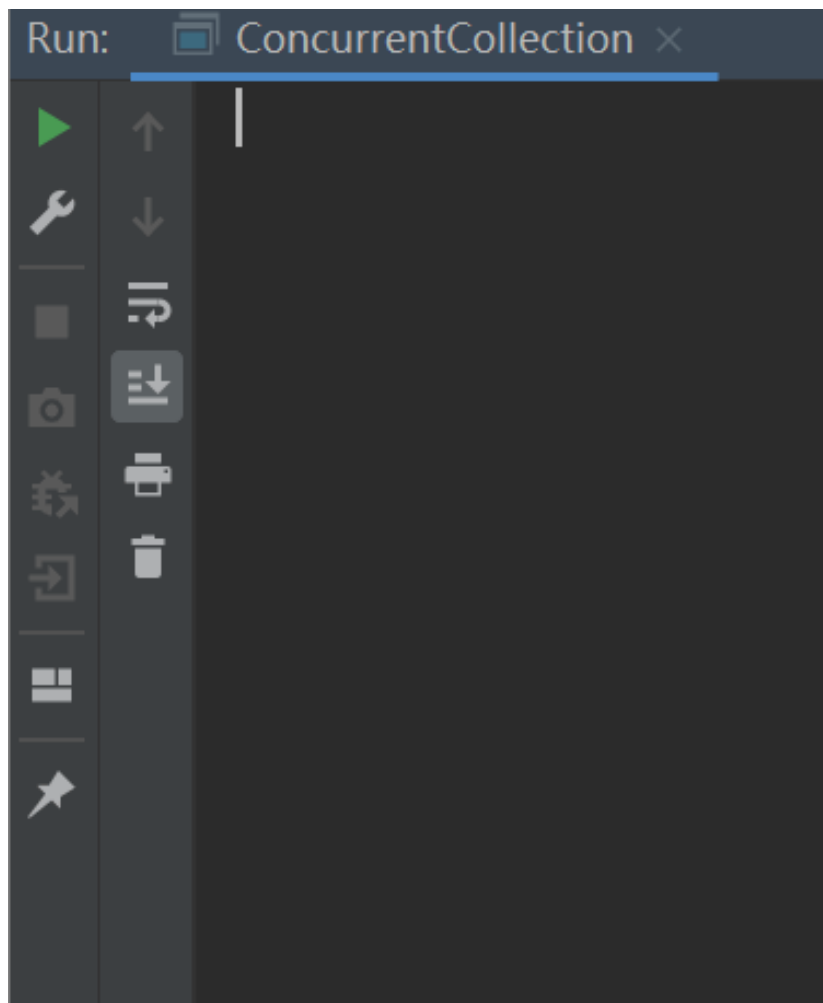
```
public static void main(String[] args) throws InterruptedException{
    List<Integer> numList = new ArrayList<>();

    Thread t1 = new Thread(new ArrayListThread(numList));
    Thread t2 = new Thread(new ArrayListThread(numList));
    Thread t3 = new Thread(new ArrayListThread(numList));
    t1.start();
    t2.start();
    t3.start();

    t1.join();
    t2.join();
    t3.join();

    System.out.println(numList.size());
}
```

# Example: Adding to ArrayList with Multithreading



```
public static void main(String[] args) throws InterruptedException {
    List<Integer> numList = new ArrayList<>();

    Thread t1 = new Thread(new ArrayListThread(numList));
    Thread t2 = new Thread(new ArrayListThread(numList));
    Thread t3 = new Thread(new ArrayListThread(numList));
    t1.start();
    t2.start();
    t3.start();

    t1.join();
    t2.join();
    t3.join();

    System.out.println(numList.size());
}
```

# Example: fail-fast iterators

```
c IteratorFailFastTest
m IteratorFailFastTest()
m runUpdateThread(): void
m runIteratorThread(): void
f list: List<Integer> = new ArrayList<>()
```

```
public void runUpdateThread() {
    Thread thread1 = new Thread(new Runnable() {
        public void run() {
            for (int i = 101; i < 200; i++) {
                list.add(i);
            }
        }
    });
    thread1.start();
}
```

```
public void runIteratorThread() {
    Thread thread2 = new Thread(new Runnable() {
        public void run() {
            ListIterator<Integer> iterator = list.listIterator();
            while (iterator.hasNext()) {
                System.out.println(iterator.next());
            }
        }
    });
    thread2.start();
}
```



# Example: fail-fast iterators

```
IteratorFailFastTest tester = new IteratorFailFastTest();  
  
tester.runIteratorThread();  
tester.runUpdateThread();
```

```
20  
21  
22  
23  
24  
Exception in thread "Thread-0" java.util.ConcurrentModificationException  
    at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:909)  
    at java.util.ArrayList$Itr.next(ArrayList.java:859)  
    at IteratorFailFastTest$2.run(IteratorFailFastTest.java:32) <1 intern
```

- Concurrent modification may lead to unexpected behavior and inconsistent results
- Fail-fast iterator prevents this by failing quickly, so that we can find and diagnose bugs early
- We should not rely on fail-fast iterator; instead, we should avoid dangerous concurrent operations

# Synchronized Wrappers

- Java Collections Framework provides factory methods for creating thread-safe collections.

```
List<String> safeList = Collections.synchronizedList(new ArrayList<>());
```

```
Map<Integer, String> unsafeMap = new HashMap<>();
```

```
Map<Integer, String> safeMap = Collections.synchronizedMap(unsafeMap);
```

- Problems?
  - Still need to explicitly put iterators in a synchronized block
  - Performance overhead: other threads need to wait for a thread to release the lock



# 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

# Copy-on-Write Collections

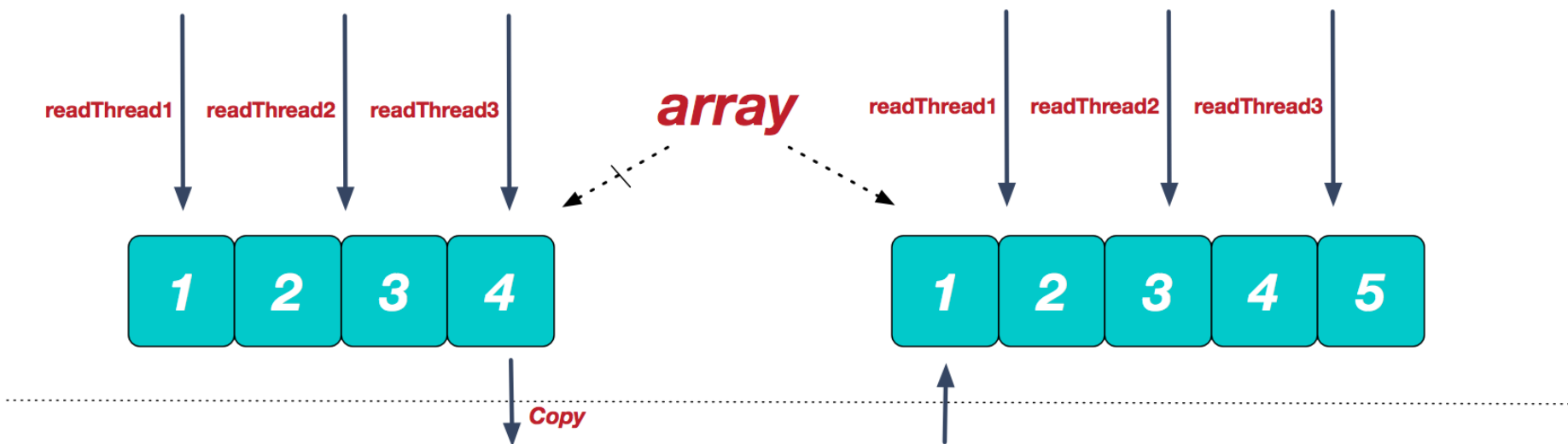
- Behaviors: sequential writes and concurrent reads
  - Reads do not block
  - Writes do not block reads, but only one write can occur at once
- Under the hood: copy-on-write collections store values in an immutable array; any change to the value of the collection results in a new array being created to reflect the new values
- Example classes
  - `CopyOnWriteArrayList`
  - `CopyOnWriteArraySet`

# 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 with some differences (checkout the official documentation or <https://www.codejava.net/java-core/concurrency/java-concurrent-collection-copyonwritearraylist-examples>)
  - AddIfAbsent(), AddAllAbsent()
  - iterator(), listIterator()

无锁并发读

将原容器引用指向新副本。  
切换过程（用volatile保证切换过程对读线程立即可见）



All modifications (add, set, remove, etc) are implemented by making a fresh copy

After modification, change the reference to the new copy

## CopyOnWriteArrayList

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



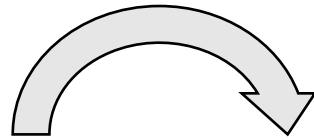
将原容器拷贝一份，写操作则作用在新副本上，需加锁。  
此过程中若有读操作则会作用在原容器上

Think: Pros & Cons?

# 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

# Collections using Lock

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



# ConcurrentHashMap

- ConcurrentHashMap added one Segment Array on top of HashMap
- Each index of the Segment array represents complete HashMap, and is guarded by a lock for put operation.

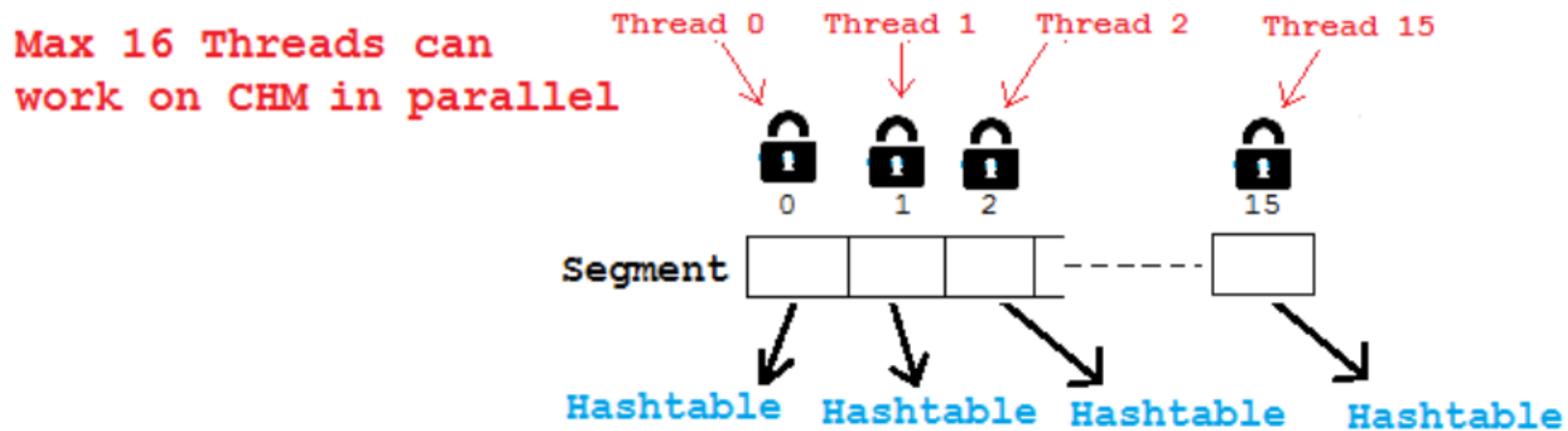
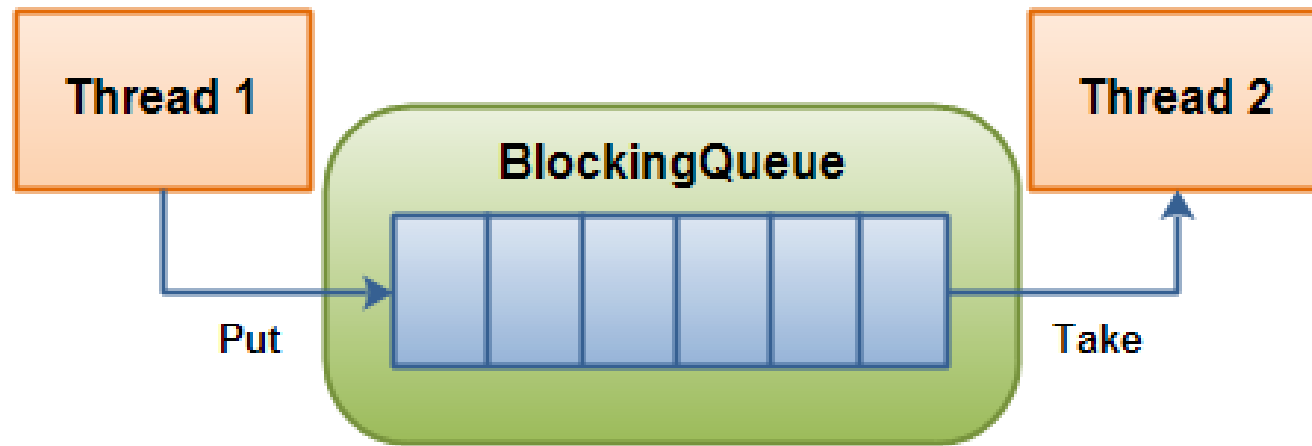


Image source: <https://javabypatel.blogspot.com/2016/09/concurrenthashmap-interview-questions.html>

# BlockingQueue

- Multiple threads can be inserting and taking elements from a BlockingQueue concurrently, without concurrency issues arising
- Typically used to have one thread produce objects (put), which another thread consumes (take)



<http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html>

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:

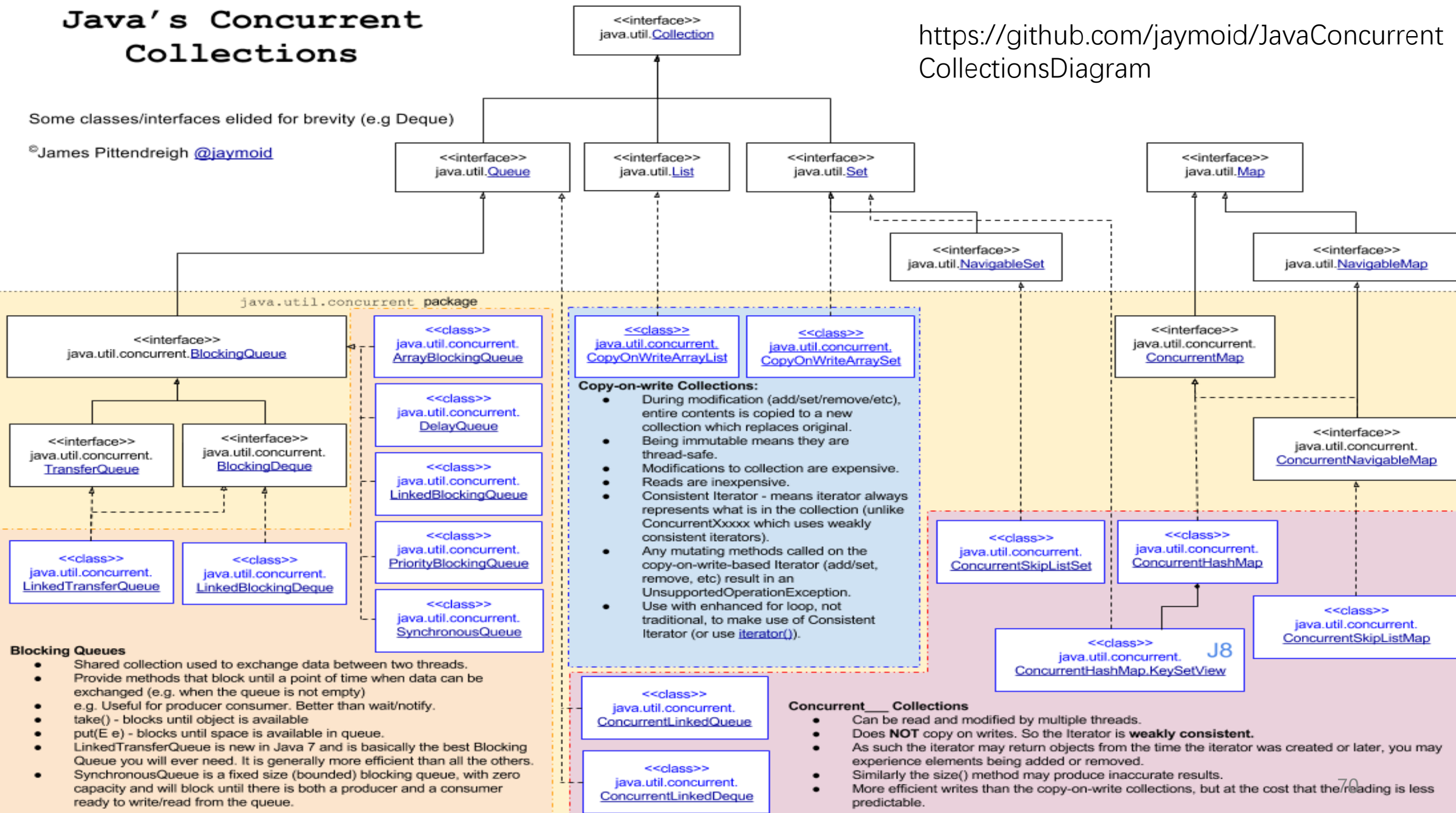
If the attempted operation is not possible immediately, the method call blocks until it is, but waits no longer than the given timeout. Returns a special value telling whether the operation succeeded or not (typically true / false).

# Java's Concurrent Collections

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

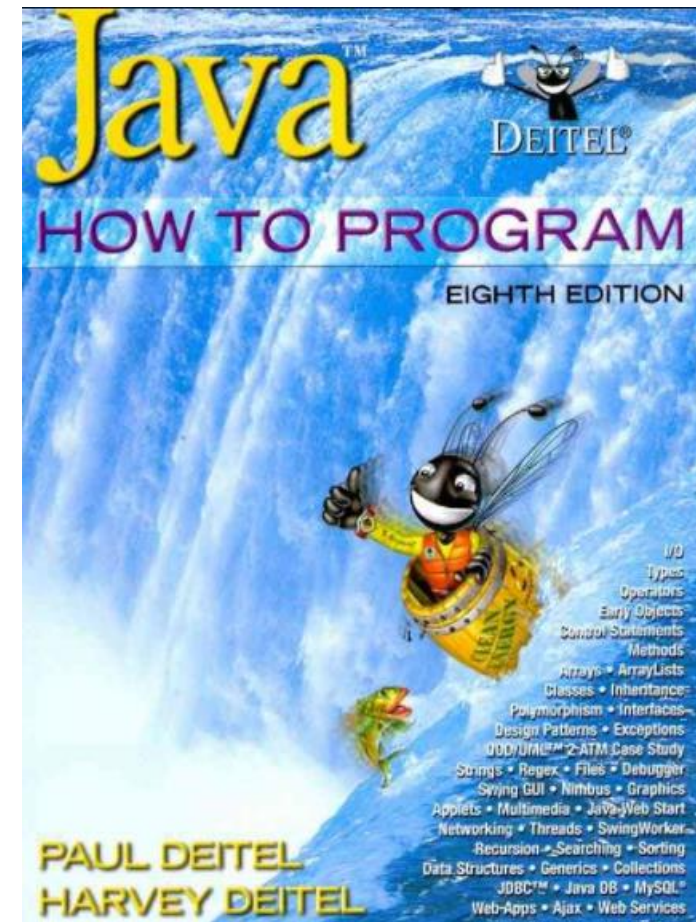
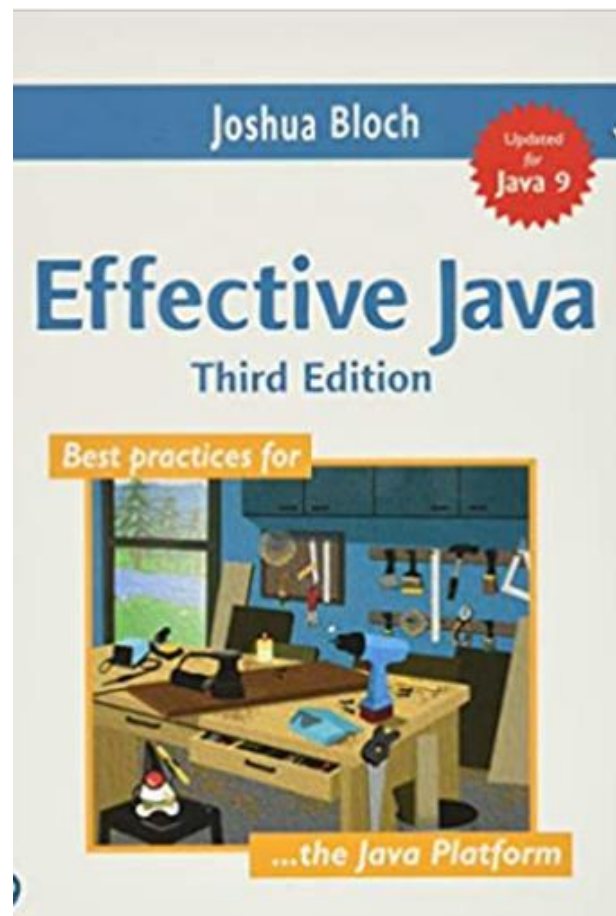
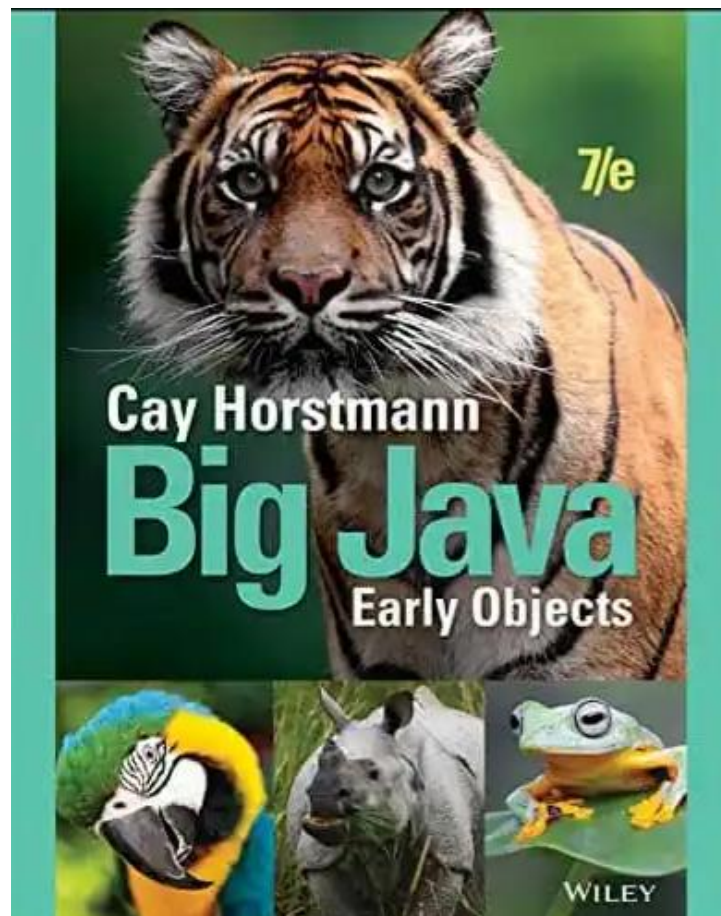
Some classes/interfaces elided for brevity (e.g Deque)

©James Pittendreigh @jaymoid





# Further Reading



# Next Lecture

- Network Programming