Java Concurrency

Ganesh Samarthyam
ganesh@codeops.tech

Do you think you always write correct concurrent programs?

Simple Program: Is it Okay?

Prints:
In run method; thread name is: main
In main method; thread name is: main

```
class MyThread extends Thread {
   public void run() {
      System.out.println("In run method; thread name is: "
        + Thread.currentThread().getName());
   }
   public static void main(String args[])  {
      Thread myThread = new MyThread();
      myThread.run();
      System.out.println("In main method; thread name is: "
        + Thread.currentThread().getName());
   }
}
```

Simple Time Bomb

Does it print "Boom!!!" after counting down from Nine to Zero? If not, how will you make it happen?

```
class TimeBomb extends Thread {
   String [] timeStr = { "Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven",
"Eight", "Nine" };

   public void run() {
      for(int i = 9; i >= 0; i--) {
         System.out.println(timeStr[i]);
      }
   }

   public static void main(String []s) {
      TimeBomb timer = new TimeBomb();
      System.out.println("Starting 10 second count down... ");
      timer.start();
      System.out.println("Boom!!!");
   }
}
```

Simple Time Bomb

In *rare* cases, it prints:
Just after creating thread;
The thread state is: NEW
Just after calling t.start();
The thread state is: TERMINATED
Just after main calling t.join();
The thread state is: TERMINATED

```
class BasicThreadStates extends Thread {
   public static void main(String []s) throws Exception {
      Thread t = new Thread(new BasicThreadStates());
      System.out.println("Just after creating thread; \n" +
           "  The thread state is: " + t.getState());
      t.start();
      System.out.println("Just after calling t.start(); \n" +
           "  The thread state is: " + t.getState());
      t.join();
      System.out.println("Just after main calling t.join(); \n" +
           "  The thread state is: " + t.getState());
   }
}
```

CodeOps

Race Condition / Data Race

```
class Counter {
   public static long count = 0;
}

class UseCounter implements Runnable {
   public static void increment() {
      Counter.count++;
      System.out.print(Counter.count + "  ");
   }
   public void run() {
      increment();
      increment();
      increment();
   }
}
```

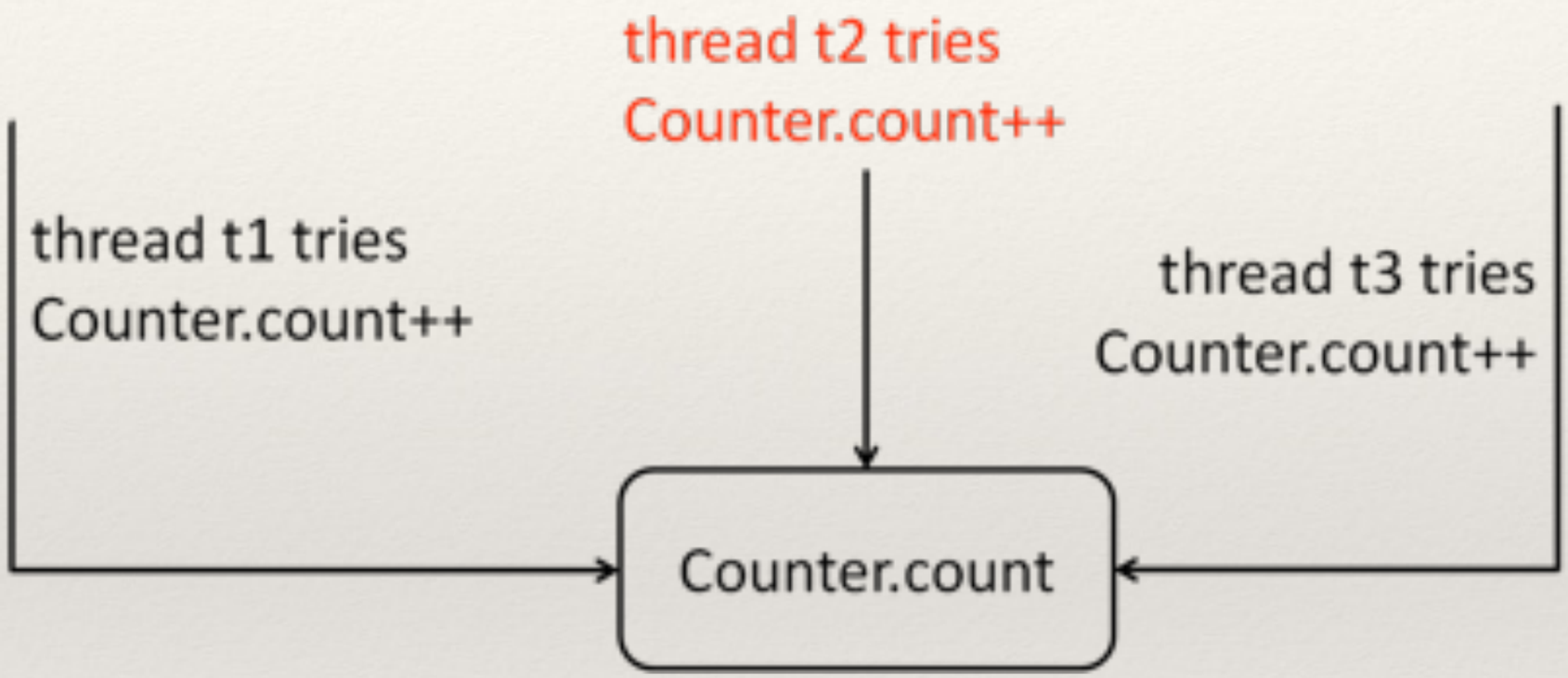
This program suffers from race condition / data race problem

```
public class DataRace {
   public static void main(String args[]) {
      UseCounter c = new UseCounter();
         Thread t1 = new Thread(c);
         Thread t2 = new Thread(c);
          Thread t3 = new Thread(c);
         t1.start();
         t2.start();
         t3.start();
   }
}
```

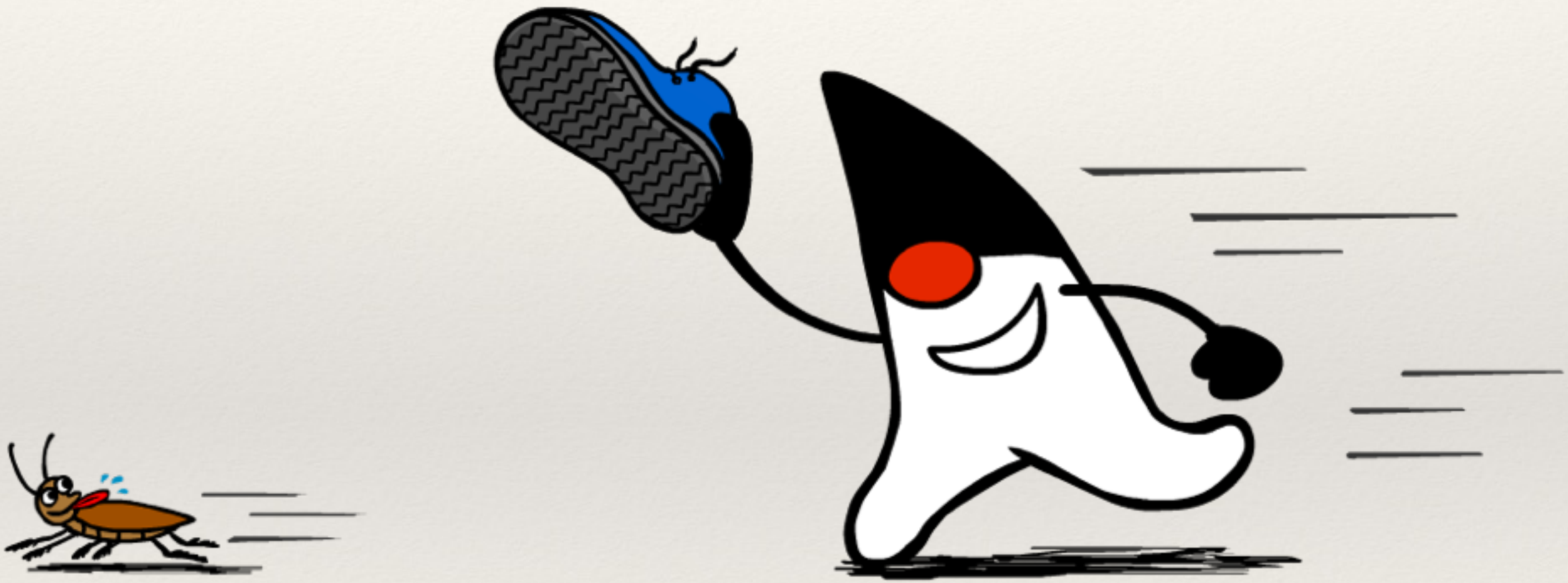
```
$ java DataRace
3  4  5  3  6  3  7  8  9
$ java DataRace
2  4  5  3  6  7  2  8  9
$ java DataRace
1  2  3  4  5  6  7  8  9
$
```

CodeOps

race conditions



Dealing with bugs in parallel programs is very difficult!



SIMPLY EXPLAINED

RACE CONDITIONS

Dead Lock

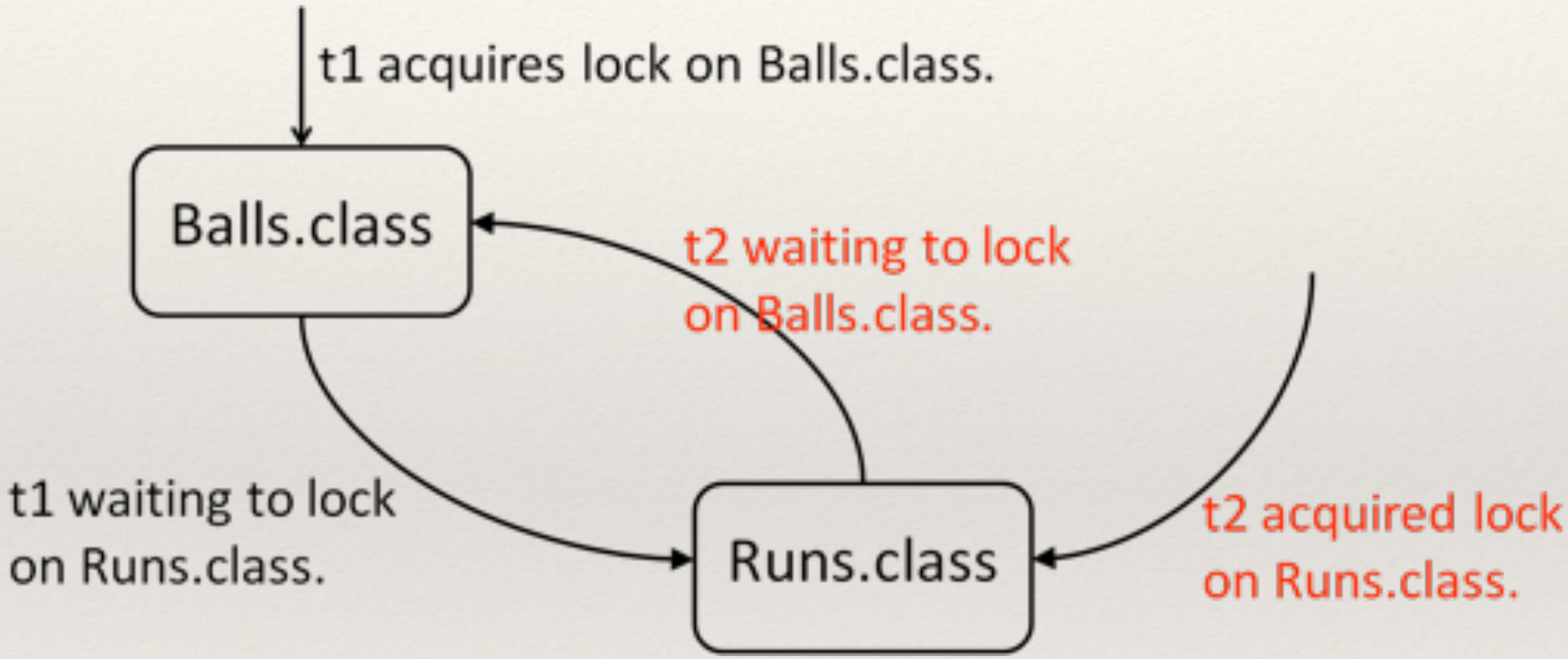
```
// Balls class has a globally accessible data member to hold the number of balls thrown
class Balls {
    public static long balls = 0;
}

// Runs class has a globally accessible data member to hold the number of runs scored
class Runs {
    public static long runs = 0;
}
```

```
public class DeadLock {
    public static void main(String args[]) throws InterruptedException {
        Counter c = new Counter();
        // create two threads and start them at the same time
        Thread t1 = new Thread(c);
        Thread t2 = new Thread(c);
        t1.start();
        t2.start();
        System.out.println("Waiting for threads to complete execution…");
        t1.join();
        t2.join();
        System.out.println("Done.");
    }
}
```

Dead Lock

```
// Counter class has two methods – IncrementBallAfterRun and IncrementRunAfterBall.
// For demonstrating deadlock, we call these two methods in the run method, so that
// locking can be requested in opposite order in these two methods
class Counter implements Runnable {
    // this method increments runs variable first and then increments the balls variable
    // since these variables are accessible from other threads,
    // we need to acquire a lock before processing them
    public void IncrementBallAfterRun() {
        // since we're updating runs variable first, first lock the Runs.class
        synchronized(Runs.class) {
            // lock on Balls.class before updating balls variable
            synchronized(Balls.class) {
                Runs.runs++;
                Balls.balls++;
            }
        }
    }

    public void IncrementRunAfterBall() {
        // since we're updating balls variable first; so first lock Balls.class
        synchronized(Balls.class) {
            // acquire lock on Runs.class before updating runs variable
            synchronized(Runs.class) {
                Balls.balls++;
                Runs.runs++;
            }
        }
    }
    public void run() {
        // call these two methods which acquire locks in different order
        // depending on thread scheduling and the order of lock acquision,
        // a deadlock may or may not arise
        IncrementBallAfterRun();
        IncrementRunAfterBall();
    }
}
```

deadlocks


t1 acquires lock on Balls.class.
Balls.class
t2 waiting to lock
on Balls.class.
t1 waiting to lock
on Runs.class.
Runs.class
t2 acquired lock
on Runs.class.

deadlocks

Too many locks!

java.util.concurrent package

Using AtomicInteger

```
import java.util.concurrent.atomic.*;

// Class to demonstrate how incrementing "normal" (i.e., thread unsafe) integers and incrementing
// "atomic" (i.e., thread safe) integers are different: Incrementing a shared Integer object without locks can result
// in a data race; however, incrementing a shared AtomicInteger will not result in a data race.

class AtomicVariableTest {
   // Create two integer objects – one normal and another atomic –  with same initial value
   private static Integer integer = new Integer(0);
   private static AtomicInteger atomicInteger = new AtomicInteger(0);

   static class IntegerIncrementer extends Thread {
      public void run() {
         ++integer;
      }
   }
   static class AtomicIntegerIncrementer extends Thread {
      public void run() {
         atomicInteger.incrementAndGet();
      }
   }
   public static void main(String []args) {
      // create three threads each for incrementing atomic and "normal" integers
      for(int i = 0; i < 10; i++) {
         new IntegerIncrementer().start();
         new AtomicIntegerIncrementer().start();
      }
      System.out.printf("final int value = %d and final AtomicInteger value = %d",
         integer, atomicInteger.intValue());
   }
}
```

Atomic Variables

- **AtomicBoolean:** Atomically updatable Boolean value.
- **AtomicInteger:** Atomically updatable int value; inherits from the N umber class.
- **AtomicIntegerArray:** An int array in which elements can be updated atomically.
- **AtomicLong:** Atomically updatable long value; inherits from N umber class.
- **AtomicLongArray:** A long array in which elements can be updated atomically.
- **AtomicReference<V>:** An atomically updatable object reference of type V.
- **AtomicReferenceArray<E>:** An atomically updatable array that can hold object references of type E (E refers to be base type of elements).

Locks

CodeOps

How to simulate only one thread using an ATM and others wait for their turn?

Locks

```
import java.util.concurrent.locks.*;

// This class simulates a situation where only one ATM machine is available and
// and five people are waiting to access the machine. Since only one person can
// access an ATM machine at a given time, others wait for their turn
class ATMMachine {
   public static void main(String []args) {
      // A person can use a machine again, and hence using a "reentrant lock"
      Lock machine = new ReentrantLock();

      // list of people waiting to access the machine
      new Person(machine, "Mickey");
      new Person(machine, "Donald");
      new Person(machine, "Tom");
      new Person(machine, "Jerry");
      new Person(machine, "Casper");
   }
}
```

Locks

```
// Each Person is an independent thread; their access to the common resource
// (the ATM machine in this case) needs to be synchronized using a lock
class Person extends Thread {
   private Lock machine;
   public Person(Lock machine, String name) {
      this.machine = machine;
      this.setName(name);
      this.start();
   }
   public void run() {
      try {
         System.out.println(getName() + " waiting to access an ATM machine");
         machine.lock();
         System.out.println(getName() + " is accessing an ATM machine");
         Thread.sleep(1000); // simulate the time required for withdrawing amount
      } catch(InterruptedException ie) {
         System.err.println(ie);
      }
      finally {
         System.out.println(getName() + " is done using the ATM machine");
         machine.unlock();
      }
   }
}
```

Locks

```
$ java ATMMachine
Donald waiting to access an ATM machine
Donald is accessing an ATM machine
Jerry waiting to access an ATM machine
Mickey waiting to access an ATM machine
Tom waiting to access an ATM machine
Casper waiting to access an ATM machine
Donald is done using the ATM machine
Jerry is accessing an ATM machine
Jerry is done using the ATM machine
Mickey is accessing an ATM machine
Mickey is done using the ATM machine
Tom is accessing an ATM machine
Tom is done using the ATM machine
Casper is accessing an ATM machine
Casper is done using the ATM machine
```

Assume that only two people can be in an ATM room at a time

Semaphore

```
import java.util.concurrent.Semaphore;

// This class simulates a situation where an ATM room has only two ATM machines
// and five people are waiting to access the machine. Since only one person can access
// an ATM machine at a given time, others wait for their turn
class ATMRoom {
   public static void main(String []args) {
      // assume that only two ATM machines are available in the ATM room
      Semaphore machines = new Semaphore(2);

      // list of people waiting to access the machine
      new Person(machines, "Mickey");
      new Person(machines, "Donald");
      new Person(machines, "Tom");
      new Person(machines, "Jerry");
      new Person(machines, "Casper");
   }
}
```

Semaphore

```
import java.util.concurrent.Semaphore;

// Each Person is an independent thread; but their access to the common resource
// (two ATM machines in the ATM machine room in this case) needs to be synchronized.
class Person extends Thread {
   private Semaphore machines;
   public Person(Semaphore machines, String name) {
      this.machines = machines;
      this.setName(name);
      this.start();
   }
   public void run() {
      try {
         System.out.println(getName() + " waiting to access an ATM machine");
         machines.acquire();
         System.out.println(getName() + " is accessing an ATM machine");
         Thread.sleep(1000); // simulate the time required for withdrawing amount
         System.out.println(getName() + " is done using the ATM machine");
         machines.release();
      } catch(InterruptedException ie) {
         System.err.println(ie);
      }
   }
}
```

Semaphore

```
Mickey waiting to access an ATM machine
Mickey is accessing an ATM machine
Jerry waiting to access an ATM machine
Jerry is accessing an ATM machine
Tom waiting to access an ATM machine
Donald waiting to access an ATM machine
Casper waiting to access an ATM machine
Mickey is done using the ATM machine
Jerry is done using the ATM machine
Tom is accessing an ATM machine
Donald is accessing an ATM machine
Tom is done using the ATM machine
Donald is done using the ATM machine
Casper is accessing an ATM machine
Casper is done using the ATM machine
```

How can you simulate a running race with a count down? Once “Start” => All threads should run

CountDownLatch

```
import java.util.concurrent.*;

// this class simulates the start of a running race by counting down from 5. It holds
// three runner threads to be ready to start in the start line of the race and once the count down
// reaches zero, all the three runners start running...

class RunningRaceStarter {
   public static void main(String []args) throws InterruptedException {
      CountDownLatch counter = new CountDownLatch(5);
      // count from 5 to 0 and then start the race

      // instantiate three runner threads
      new Runner(counter, "Carl");
      new Runner(counter, "Joe");
      new Runner(counter, "Jack");

      System.out.println("Starting the countdown ");
      long countVal = counter.getCount();
      while(countVal > 0) {
         Thread.sleep(1000); // 1000 milliseconds = 1 second
         System.out.println(countVal);
         if(countVal == 1) {
            // once counter.countDown(); in the next statement is called,
            // Count down will reach zero; so shout "Start"
            System.out.println("Start");
         }
         counter.countDown(); // count down by 1 for each second
         countVal = counter.getCount();
      }
   }
}
```

CountDownLatch

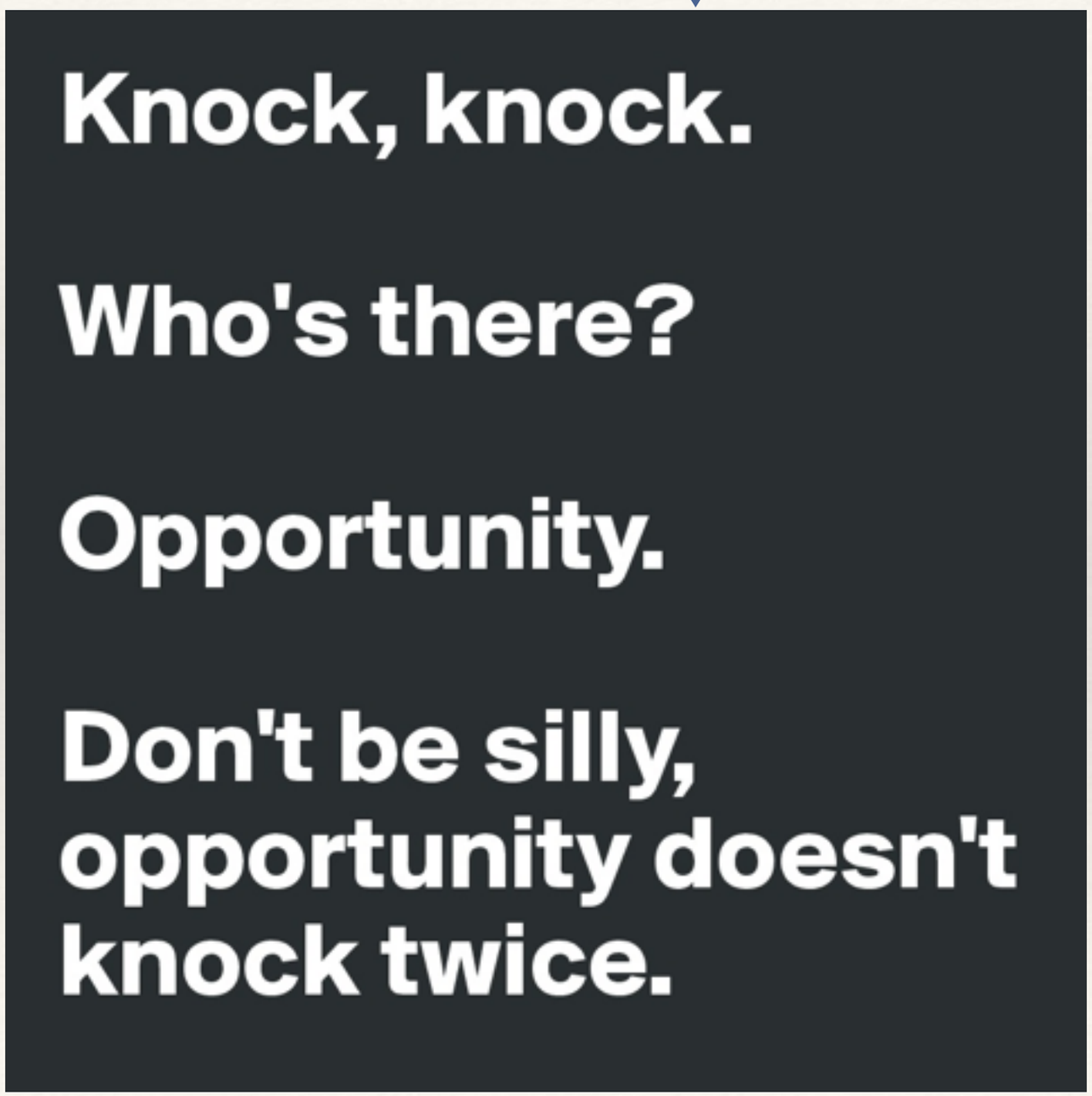
```
// this Runner class simulates a track runner in a 100-meter dash race. The runner waits till the
// count down timer gets to zero and then starts running
class Runner extends Thread {
   private CountDownLatch timer;
   public Runner(CountDownLatch cdl, String name) {
      timer = cdl;
      this.setName(name);
      System.out.println(this.getName() + " ready and waiting for the count down to start");
      start();
   }

   public void run() {
      try {
         // wait for the timer count down to reach 0
         timer.await();
      } catch (InterruptedException ie) {
         System.err.println("interrupted -- can't start running the race");
      }
      System.out.println(this.getName() + " started running");
   }
}
```

CodeOps

CountDownLatch

```
Carl ready and waiting for the count down to start
Joe ready and waiting for the count down to start
Jack ready and waiting for the count down to start
Starting the countdown
5
4
3
2
1
Start
Carl started running
Jack started running
Joe started running
```

How to exchange data (or synchronise communication) between two threads?



Exchanger

```
class CoffeeShopThread extends Thread {
        private Exchanger<String> sillyTalk;

        public CoffeeShopThread(Exchanger<String> args) {
                sillyTalk = args;
        }
        public void run() {
                String reply = null;
                try {
                        // exchange the first messages
                        reply = sillyTalk.exchange("Who's there?");
                        // print what Duke said
                        System.out.println("Duke: " + reply);

                        // exchange second message
                        reply = sillyTalk.exchange("Duke who?");
                        // print what Duke said
                        System.out.println("Duke: " + reply);

                        // there is no message to send, but to get a message from Duke thread,
                        // both ends should send a message; so send a "dummy" string
                        reply = sillyTalk.exchange("");
                        System.out.println("Duke: " + reply);
                } catch(InterruptedException ie) {
                        System.err.println("Got interrupted during my silly talk");
                }
        }
}
```

Exchanger

```
import java.util.concurrent.Exchanger;

// The DukeThread class runs as an independent thread. It talks to the CoffeeShopThread that
// also runs independently. The chat is achieved by exchanging messages through a common
// Exchanger<String> object that synchronizes the chat between them.
// Note that the message printed are the "responses" received from CoffeeShopThread
class DukeThread extends Thread {
        private Exchanger<String> sillyTalk;

        public DukeThread(Exchanger<String> args) {
                sillyTalk = args;
        }
        public void run() {
                String reply = null;
                try {
                        // start the conversation with CoffeeShopThread
                        reply = sillyTalk.exchange("Knock knock!");
                        // Now, print the response received from CoffeeShopThread
                        System.out.println("CoffeeShop: " + reply);

                        // exchange another set of messages
                        reply = sillyTalk.exchange("Duke");
                        // Now, print the response received from CoffeeShopThread
                        System.out.println("CoffeeShop: " + reply);

                        // an exchange could happen only when both send and receive happens
                        // since this is the last sentence to speak, we close the chat by
                        // ignoring the "dummy" reply
                        reply = sillyTalk.exchange("The one who was born in this coffee shop!");
                        // talk over, so ignore the reply!
                } catch(InterruptedException ie) {
                        System.err.println("Got interrupted during my silly talk");
                }
        }
}
```

Exchanger

```
// Co-ordinate the silly talk between Duke and CoffeeShop by instantiating the Exchanger object
// and the CoffeeShop and Duke threads
class KnockKnock {
        public static void main(String []args) {
                Exchanger<String> sillyTalk = new Exchanger<String>();
                new CoffeeShopThread(sillyTalk).start();
                new DukeThread(sillyTalk).start();
        }
}
```

Exchanger

```
CoffeeShop: Who's there?
Duke: Knock knock!
Duke: Duke
CoffeeShop: Duke who?
Duke: The one who was born in this coffee shop!
```

Useful Concurrency Utilities

- **Semaphore:** Controls access to one or more shared resources
- **Phaser:** Supports a synchronization barrier
- **CountDownLatch:** Allows threads to wait for a countdown to complete
- **Exchanger:** Supports exchanging data between two threads
- **CyclicBarrier:** Enables threads to wait at a predefined execution point

Concurrent Collections

Class/Interface	Short Description
`BlockingQueue`	This interface extends the `Queue` interface. In `BlockingQueue`, if the queue is empty, it waits (i.e., blocks) for an element to be inserted, and if the queue is full, it waits for an element to be removed from the queue.
`ArrayBlockingQueue`	This class provides a fixed-sized array based implementation of the `BlockingQueue` interface.
`LinkedBlockingQueue`	This class provides a linked-list-based implementation of the `BlockingQueue` interface.
`DelayQueue`	This class implements `BlockingQueue` and consists of elements that are of type `Delayed`. An element can be retrieved from this queue only after its delay period.
`PriorityBlockingQueue`	Equivalent to `java.util.PriorityQueue`, but implements the `BlockingQueue` interface.
`SynchronousQueue`	This class implements `BlockingQueue`. In this container, each `insert`() by a thread waits (blocks) for a corresponding `remove`() by another thread and vice versa.
`LinkedBlockingDeque`	This class implements `BlockingDeque` where insert and remove operations could block; uses a linked-list for implementation.
`ConcurrentHashMap`	Analogous to `Hashtable`, but with safe concurrent access and updates.
`ConcurrentSkipListMap`	Analogous to `TreeMap`, but provides safe concurrent access and updates.
`ConcurrentSkipListSet`	Analogous to `TreeSet`, but provides safe concurrent access and updates.
`CopyOnWriteArrayList`	Similar to `ArrayList`, but provides safe concurrent access. When the `ArrayList` is updated, it creates a fresh copy of the underlying array.
`CopyOnWriteArraySet`	A `Set` implementation, but provides safe concurrent access and is implemented using `CopyOnWriteArrayList`. When the container is updated, it creates a fresh copy of the underlying array.

Priority Queue Doesn't "Block"

```
import java.util.*;

// Simple PriorityQueue example. Here, we create two threads in which one thread inserts an element,
// and another thread removes an element from the priority queue.
class PriorityQueueExample {
   public static void main(String []args) {
      final PriorityQueue<Integer> priorityQueue = new PriorityQueue<>();
      // spawn a thread that removes an element from the priority queue
      new Thread() {
         public void run() {
            // Use remove() method in PriorityQueue to remove the element if available
            System.out.println("The removed element is: " + priorityQueue.remove());
         }
      }.start();
      // spawn a thread that inserts an element into the priority queue
      new Thread() {
         public void run() {
            // insert Integer value 10 as an entry into the priority queue
            priorityQueue.add(10);
            System.out.println("Successfully added an element to the queue ");
         }
      }.start();
   }
}
```

Priority Queue Doesn't "Block"

```
java PriorityQueueExample
Exception in thread "Thread-0"
java.util.NoSuchElementException
   at java.util.AbstractQueue.remove(AbstractQueue.java:117)
   at PriorityQueueExample$1.run(PriorityQueueExample.java:12)
Successfully added an element to the queue
```

Using PriorityBlockingQueue

```
// Illustrates the use of PriorityBlockingQueue. In this case, if there is no element available in the priority queue
// the thread calling take() method will block (i.e., wait) till another thread inserts an element

import java.util.concurrent.*;

class PriorityBlockingQueueExample {
    public static void main(String []args) {
        final PriorityBlockingQueue<Integer> priorityBlockingQueue = new PriorityBlockingQueue<>();
        new Thread() {
            public void run() {
                try {
                // use take() instead of remove()
                 // note that take() blocks, whereas remove() doesn't block
                System.out.println("The removed element is: " + priorityBlockingQueue.take());
            } catch(InterruptedException ie) {
                // its safe to ignore this exception
                ie.printStackTrace();
            }
        }
    }.start();
    new Thread() {
        public void run() {
            // add an element with value 10 to the priority queue
            priorityBlockingQueue.add(10);
            System.out.println("Successfully added an element to the que);
    }
}
```

Using PriorityBlockingQueue

```
$ java PriorityBlockingQueueExample
Successfully added an element to the queue
The removed element is: 10
```

Modifying a List Concurrently

```
import java.util.*;

public class ModifyingList {
    public static void main(String []args) {
        List<String> aList = new ArrayList<>();
        aList.add("one");
        aList.add("two");
        aList.add("three");

        Iterator listIter = aList.iterator();
        while(listIter.hasNext()) {
            System.out.println(listIter.next());
            aList.add("four");
        }
    }
}
```

```
$ java ModifyingList
one
Exception in thread "main" java.util.ConcurrentModificationException
   at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:901)
   at java.util.ArrayList$Itr.next(ArrayList.java:851)
   at ModifyingList.main(ModifyingList.java:14)
```

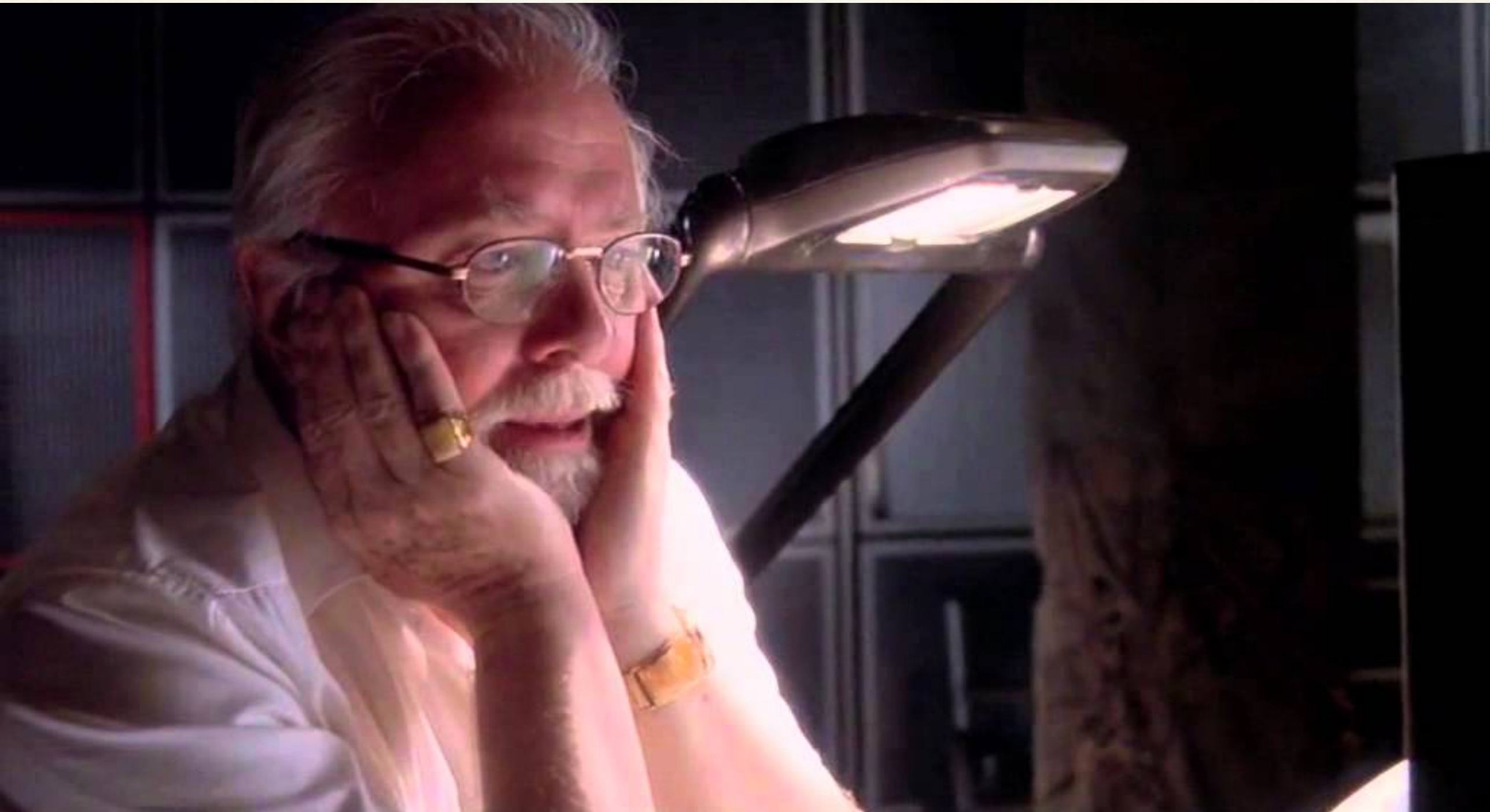
CopyOnWriteArrayList

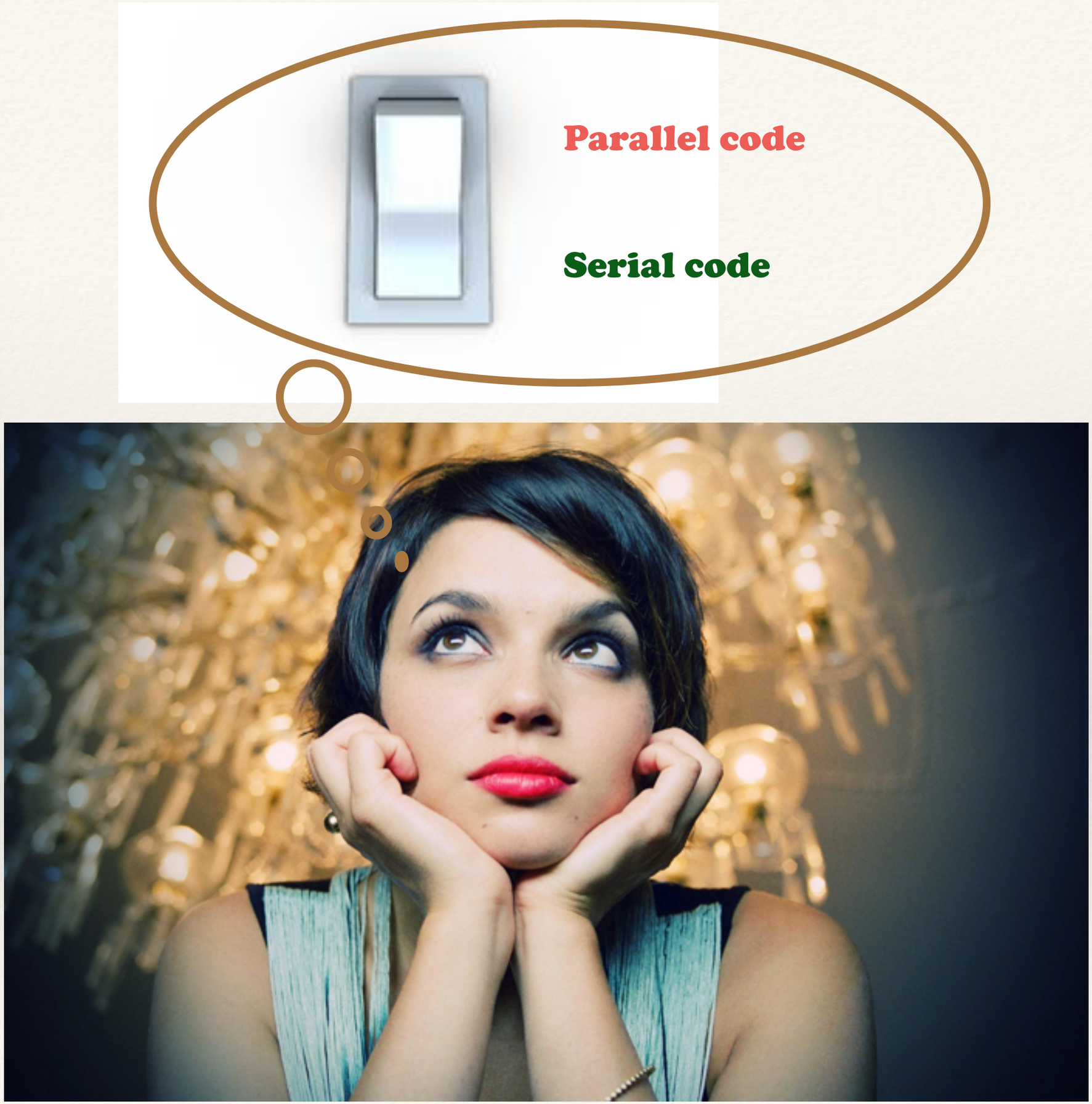
```
import java.util.*;
import java.util.concurrent.CopyOnWriteArrayList;

public class COWList {
    public static void main(String []args) {
        List<String> aList = new CopyOnWriteArrayList<>();
        aList.add("one");
        aList.add("two");
        aList.add("three");

        Iterator listIter = aList.iterator();
        while(listIter.hasNext()) {
            System.out.println(listIter.next());
            aList.add("four");
        }
    }
}
```

```
$ java COWList
one
two
three
```

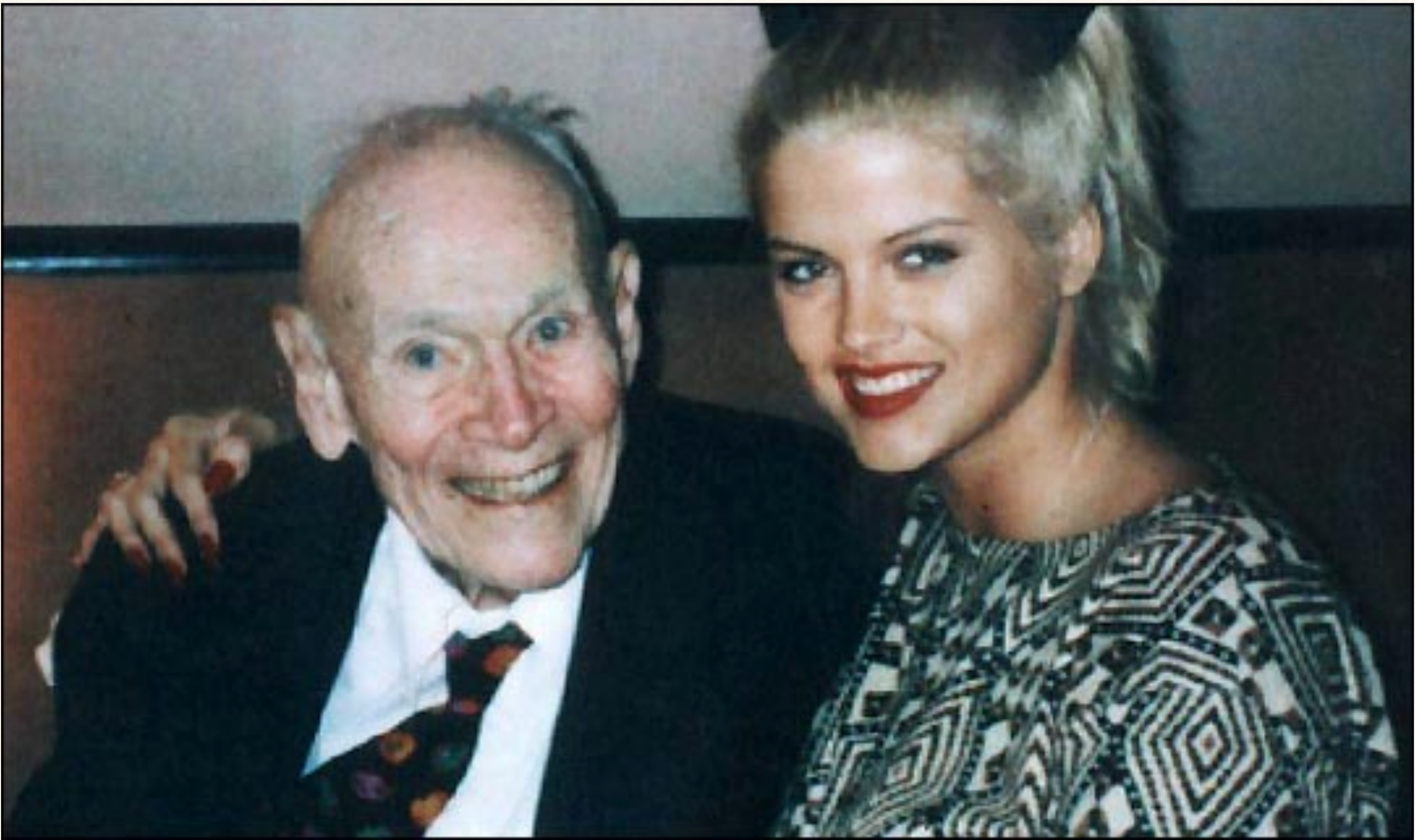
Parallel Streams

I really really hate concurrency problems




Parallel code
Serial code

Sometimes, dreams do come true even at 86 :-)




So, keep dreaming till you become 86!

Parallel Streams

```
import java.util.stream.LongStream;

class PrimeNumbers {
   private static boolean isPrime(long val) {
      for(long i = 2; i <= val/2; i++) {
         if((val % i) == 0) {
            return false;
         }
      }
      return true;
   }
   public static void main(String []args) {
      long numOfPrimes = LongStream.rangeClosed(2, 50_000)
                .parallel()
                .filter(PrimeNumbers::isPrime)
                .count();
      System.out.println(numOfPrimes);
   }
}
```

Prints 9592

```
long numOfPrimes = LongStream.rangeClosed(2, 100_000)
                             .filter(PrimeNumbers::isPrime)
                             .count();
System.out.println(numOfPrimes);
```

2.510 seconds

Let’s flip the switch by calling parallel() function

Parallel code

Serial code

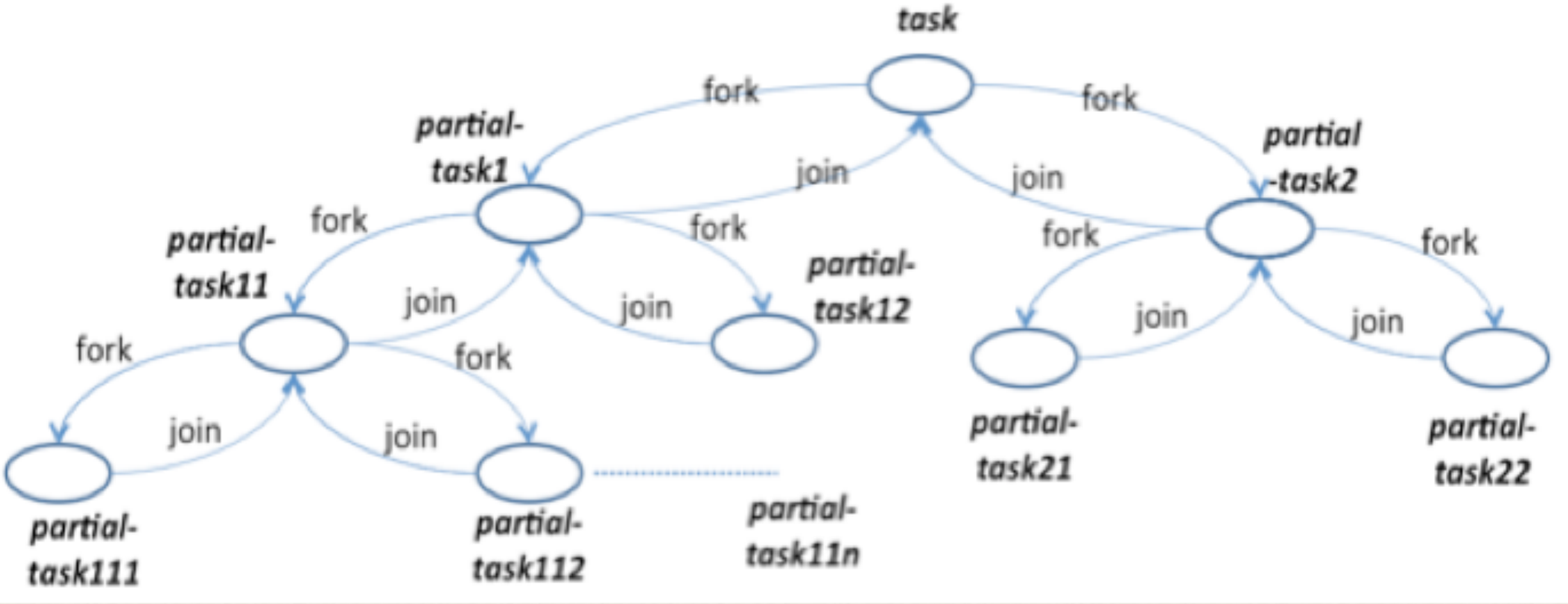
Prints 9592

```
long numOfPrimes = LongStream.rangeClosed(2, 100_000)
                             .parallel()
                             .filter(PrimeNumbers::isPrime)
                             .count();
System.out.println(numOfPrimes);
```

1.235 seconds


Wow! That's an awesome flip switch!

Internally, parallel streams make use of fork-join framework



CodeOps


WATCH
FOR
ICE

Gives wrong results with with parallel() call

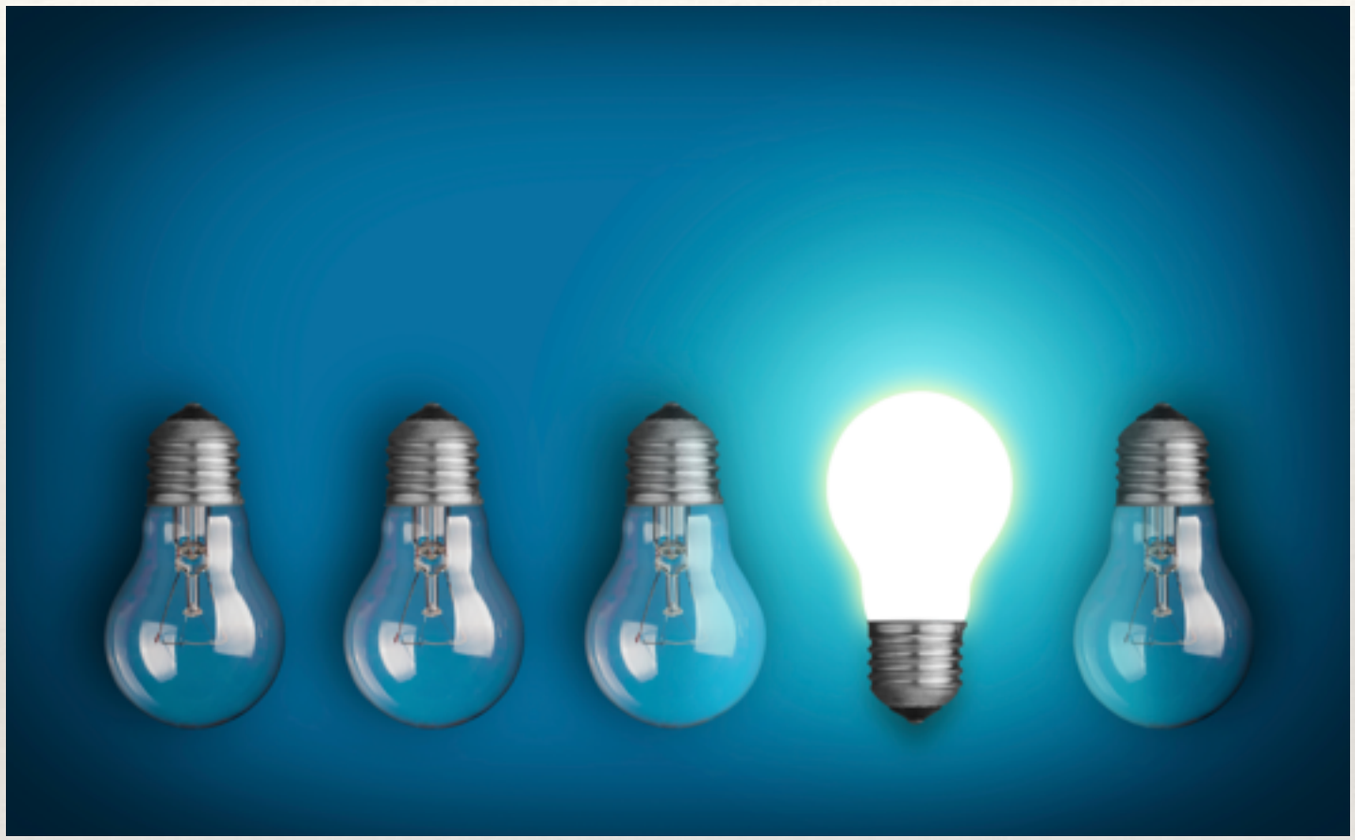
```
import java.util.Arrays;

class StringConcatenator {
    public static String result = "";
    public static void concatStr(String str) {
        result = result + " " + str;
    }
}

class StringSplitAndConcatenate {
    public static void main(String []args) {
        String words[] = "the quick brown fox jumps over the lazy dog".split(" ");
        Arrays.stream(words).forEach(StringConcatenator::concatStr);
        System.out.println(StringConcatenator.result);
    }
}
```

Meetups

http://www.meetup.com/JavaScript-Meetup-Bangalore/
http://www.meetup.com/Container-Developers-Meetup-Bangalore/
http://www.meetup.com/Software-Craftsmanship-Bangalore-Meetup/
http://www.meetup.com/Core-Java-Meetup-Bangalore/
http://www.meetup.com/Technical-Writers-Meetup-Bangalore/
http://www.meetup.com/CloudOps-Meetup-Bangalore/
http://www.meetup.com/Bangalore-SDN-IoT-NetworkVirtualization-Enthusiasts/
http://www.meetup.com/SoftwareArchitectsBangalore/

CodeOps

Upcoming Bootcamps



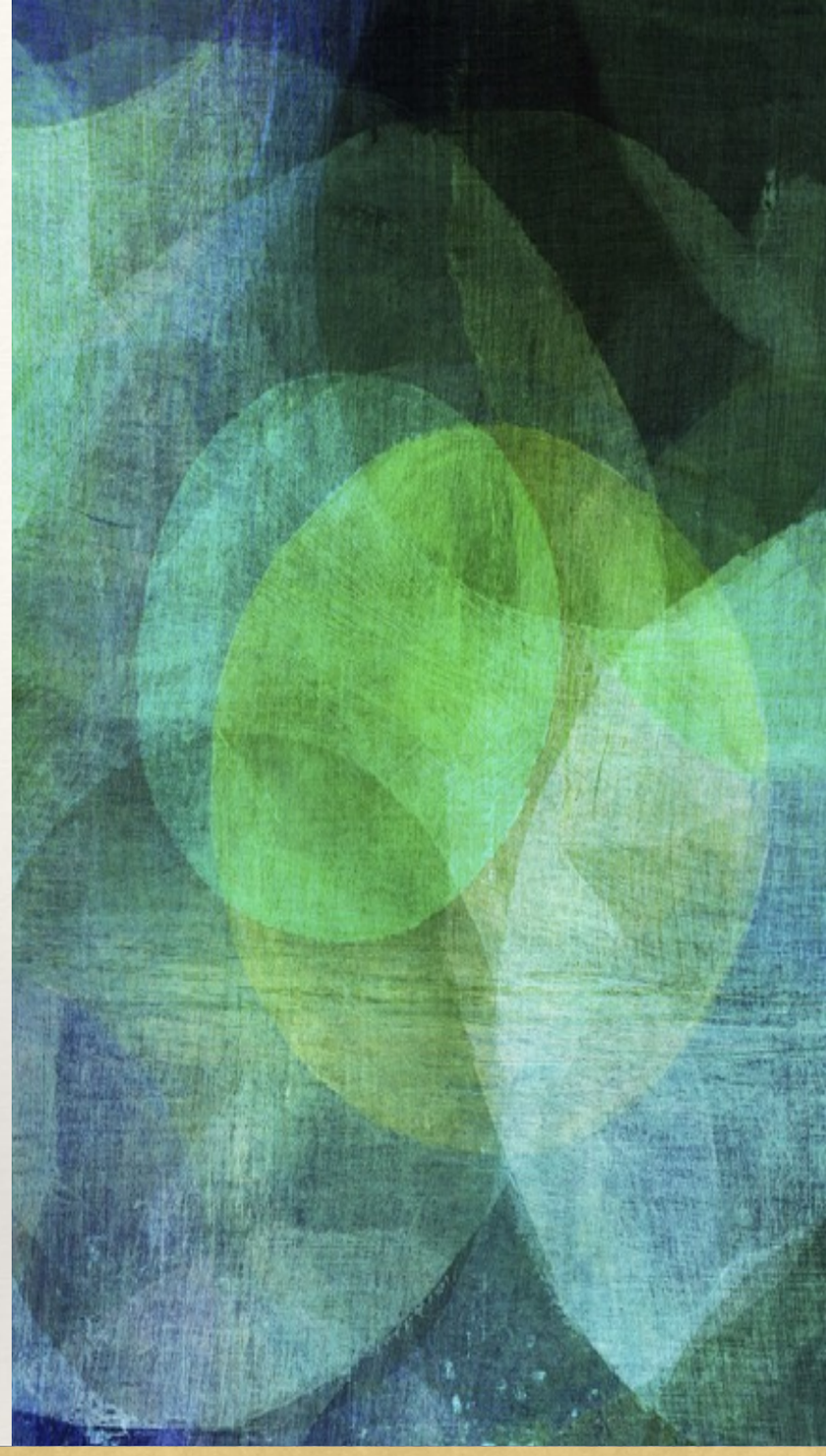
Apply coupon code "FLAT500" to get Rs. 500 discount

https://www.townscript.com/e/solidprinciples (27th August)

https://www.townscript.com/e/azure (27th August)

https://www.townscript.com/e/modernarchitecture (10th Sept)

CodeOps



ganesh@codeops.tech

www.codeops.tech

+91 98801 64463

@GSamarthyam

slideshare.net/sgganesh

bit.ly/ganeshsg

CodeOps

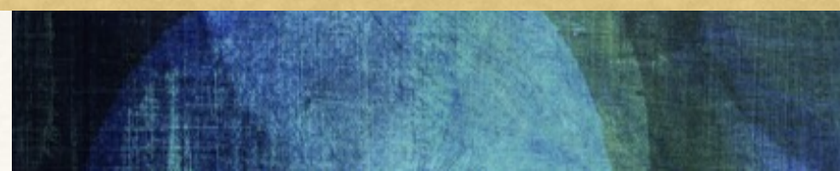


Image Credits

- https://bonexpose.com/wp-content/uploads/2015/07/Wildlife-Photography-by-Tim-Flach-1a.jpg
- http://worldofdtcmarketing.com/wp-content/uploads/2016/02/innovation-image1111111.jpg
- https://d.ibtimes.co.uk/en/full/1392314/traffic-light-tree.jpg
- http://s3-media1.fl.yelpcdn.com/bphoto/BKYrqfRerQFqM1wQWtYphw/o.jpg
- http://popsych.org/wp-content/uploads/2013/03/starting-line.jpeg
- https://cdn.boldomatic.com/content/post/419493-05f182bf5bb701ed8ac64c963395eade77b07a674117a0dadba2a5f1404eca5f/Knock-knock-Who-s-there-Opportunity-Don-t-be-silly?size=800
- http://www.locksmith-in-huddersfield.com/commercial_locks.png
- https://www.discoveryindochina.com/fr/wp-content/uploads/454.jpg