Advanced Object-Oriented Programming

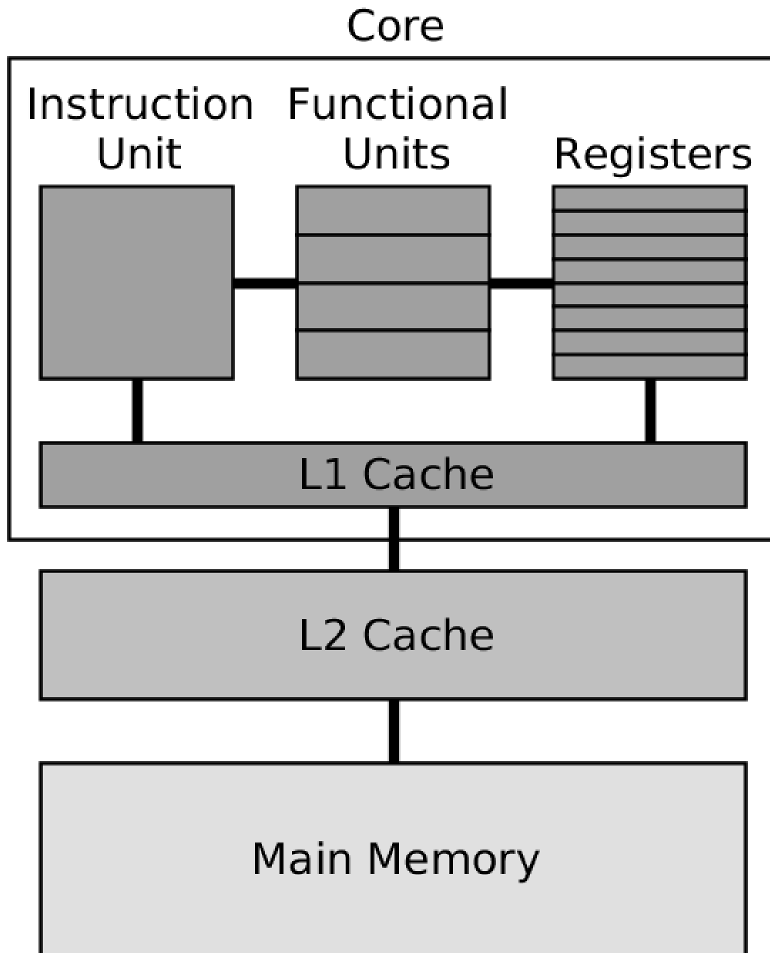
Java Concurrency

Dr. Kulwadee Somboonviwat

International College, KMITL

kskulwad@kmitl.ac.th

Before 2004: a Single core computer

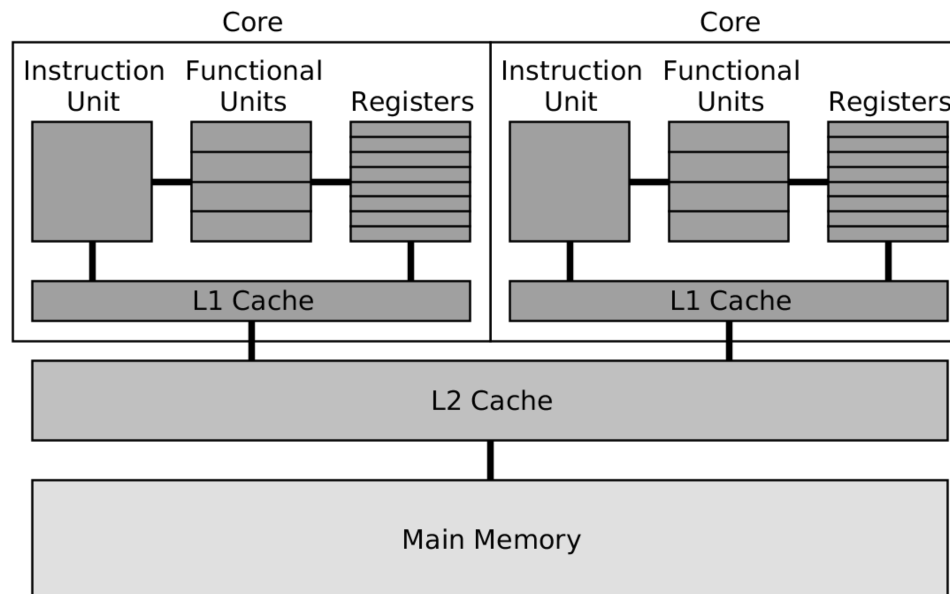


A ***core*** refers to the hardware on a computer that executes a stream of machine instruction. The core consists of:

- **Instruction unit:** read-decode-execute program's machine instructions
- **Functional units:** carries out machine instructions (e.g. shifters, adders, multipliers)
- **Registers:** high speed memory for keeping intermediate results

2004 onwards: Multi-core computers…

- Before 2004 (single-core)
  - increase the processing power by increasing clock frequencies
    - Issue: power dissipation → heat
- Multi-core processors
  - increase the processing power by adding more cores



Process

- An instance of **a program that is being executed**
- Provides an execution environment
- In general, a process consists of:
    - Executable machine code image
    - Memory: code, data, call stack, heap
    - Resource descriptors e.g. file descriptors
    - Security attributes
    - Processor state (*context*) e.g. registers content

Thread

- The smallest sequence of programmed instructions that can be managed independently by an OS
- Thread exists within a process
  - every process has at least one thread
- Multiple threads shares the process's resources, including memory and open files.
- Multithreaded execution is an essential feature of the Java platform.

Multithreaded programming in Java

- Java built-in support for concurrent programming
  - Low level concurrency constructs
    - Thread and Runnable
    - Synchronized keyword and implicit locks
  - High level concurrency constructs
    - the java.util.concurrency package
    - Synchronizer classes e.g. Semaphore, Exchanger
    - Thread-safe collections
    - Atomic Variables, Locks and conditions
    - Executors and ThreadPools
    - Parallel Fork/Join

Thread Objects

- In Java, each thread is associated with an instance of the class Thread. A thread can be created in two ways:
  - Extending the Thread class
  - Implementing the Runnable interface

Extending the Thread class

```
class MyThread1 extends Thread {
        public void run() {
                try {
                        sleep(1000);
                }
                catch (InterruptedException ex) {
                         ex.printStackTrace();
                         // ignore the InterruptedException - this is perhaps the one of the
                         // very few of the exceptions in Java which is acceptable to ignore
                }
                System.out.println("In run method; thread name is: "+getName());
        }
        public static void main(String args[])  {
                Thread myThread=new MyThread1();
                myThread.start();
                System.out.println("In main method; thread name is: "+
                         Thread.currentThread().getName());
        }
}
```

Implementing the Runnable interface

```
class MyThread2 implements Runnable {
        public void run() {
                System.out.println("In run method; thread name is: "+
                Thread.currentThread().getName());
        }

        public static void main(String args[]) throws Exception {
                Thread myThread=new Thread(new MyThread2());
                myThread.start();
                System.out.println("In main method; thread name is: "+
                Thread.currentThread().getName());
        }
}
```

Pausing a Thread with Thread.sleep()

```
class TimeBombNoJoin implements Runnable {
    public void run() {
        for (int i=9; i>=0; i--) {
            try {
                System.out.println(i);
                Thread.currentThread().sleep(1000);
            }
            catch (InterruptedException ie) {
                ie.printStackTrace();
            }
        }
    }

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

```
Starting 10 second count down...
Boom!!
9
8
7
6
5
4
3
2
1
0
```

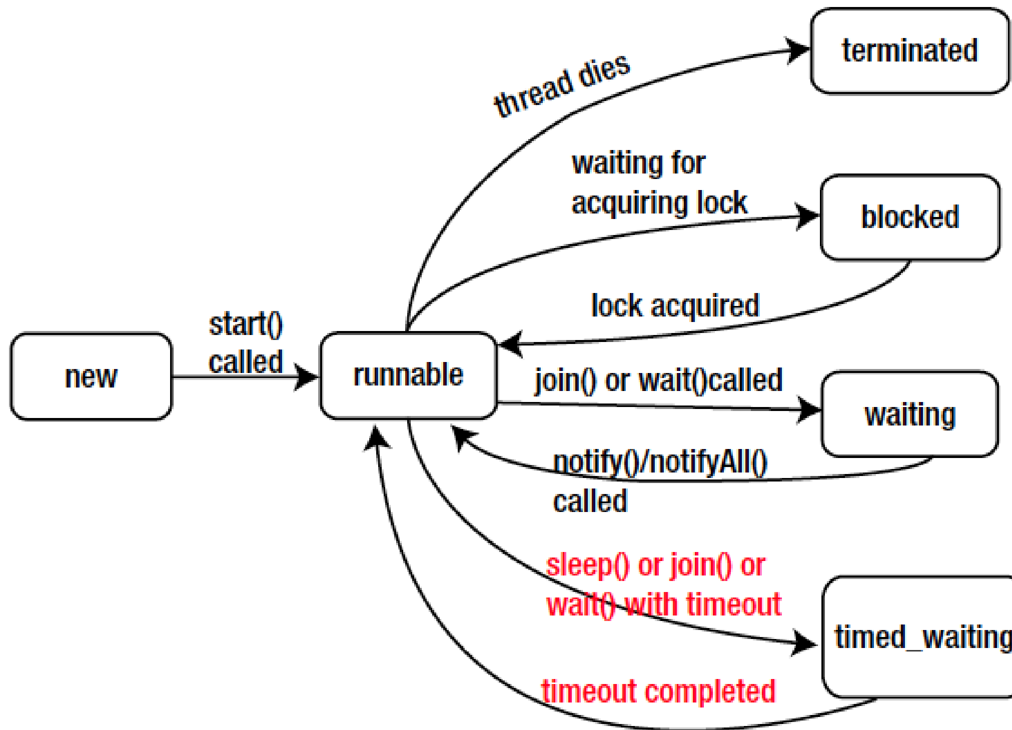
Pausing a Thread with Thread.join()

```java
class TimeBomb implements Runnable {
    public void run() {
        for (int i=9; i>=0; i--) {
            try {
                System.out.println(i);
                Thread.currentThread().sleep(1000);
            }
            catch (InterruptedException ie) {
                ie.printStackTrace();
            }
        }
    }

    public static void main(String[] args) {
        Thread timerThread = new Thread(new TimeBomb());
        System.out.println("Starting 10 second count down...");
        timerThread.start();
        try {
            timerThread.join(); // waiting for the timerThread to terminate
        }
        catch (InterruptedException ie) {
            ie.printStackTrace();
        }
        System.out.println("Boom!!");
    }
}
```

```
Starting 10 second count down...
9
8
7
6
5
4
3
2
1
0
Boom!!
```

The States of a Thread: Thread.State enum




```
class ThreadStatesEnumeration {
        public static void main(String []s) {
                for(Thread.State state : Thread.State.values()){
                        System.out.println(state);
                }
        }
}
```

Concurrent Access Problems

- Two key problems
  - Data races
  - Deadlocks

Data Races

- Threads share memory, and they can concurrently modify data. Since the modification can be done at the same time without safeguards, this can lead to unintuitive results.
- When two or more threads are trying to access a variable and one of them wants to modify it, you get a problem known as a **data race** (also called as **race condition** or **race hazard**).

Data races

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

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

public class DataRace {
    public static void main(String[] args) throws Exception {
        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();
    }
}
```

1st run:

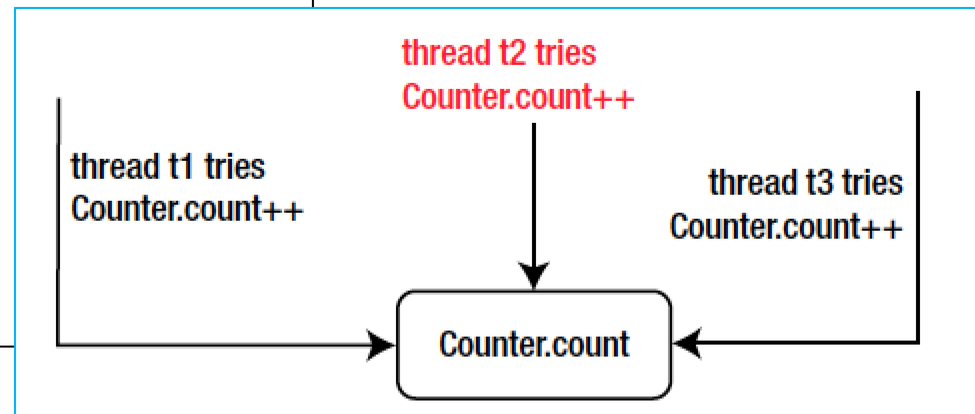
2 3 4 2 6 7 5 8 9

2nd run:

1 2 3 5 4 7 8 6 9

3rd run:

3 3 5 6 3 7 8 4 9



Thread Synchronization

- The section of code that is accessed and modified by more than one threads is called **critical section**
- To avoid the data race problem, we need to ensure that the critical section is executed by only one thread at a time
    - Use a **lock** to protect the critical section
    - Only a single thread can acquire a lock on an object at a time, and only that thread can execute the block of code
- Java support for **locking/unlocking:**
    - **Implicit lock** : the **synchronized** keyword
    - **Explicit lock** : the **java.util.concurrent.locks.Lock** interface

Synchronized methods / blocks

// synchorized method

```
public synchronized void increment() {
    Counter.count++;
    System.out.print(Counter.count + " ");
}
```

// synchorized block

```
public void increment() {
    synchronized(this) {
        Counter.count++;
        System.out.print(Counter.count + " ");
    }
}
```

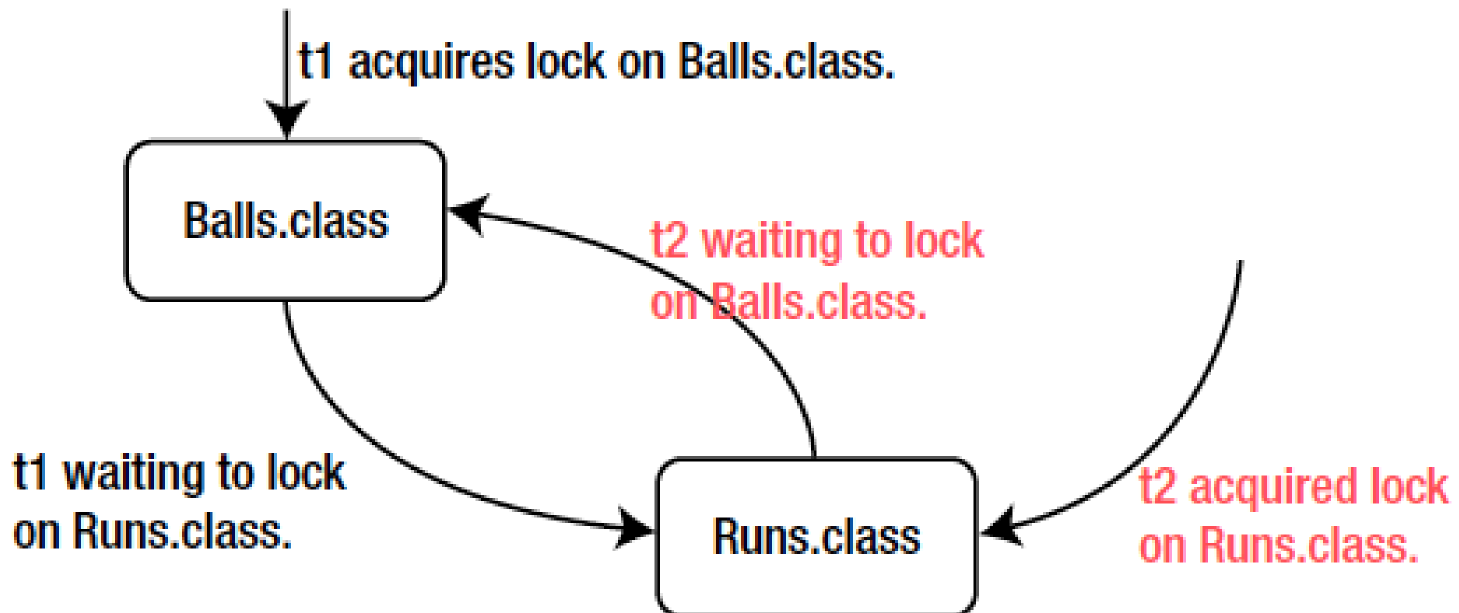
```
$ java DataRaceSolved
1 2 3 4 5 6 7 8 9
```

DeadLock.java

```
class Balls { public static long balls = 0; }
class Runs { public static long runs = 0; }
class Counter implements Runnable {
    public void IncrementBallAfterRun() {
        synchronized(Runs.class) {
            synchronized(Balls.class) {
                Runs.runs++;
                Balls.balls++;
            }
        }
    }
    public void IncrementRunAfterBall() {
        synchronized(Balls.class) {
            synchronized(Runs.class) {
                Balls.balls++;
                Runs.runs++;
            }
        }
    }
    public void run() {
        IncrementBallAfterRun();
        IncrementRunAfterBall();
    }
}
public class DeadLock {
    public static void main(String[] args) throws InterruptedException {
        Counter c = new Counter();
        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.");
    }
}
```

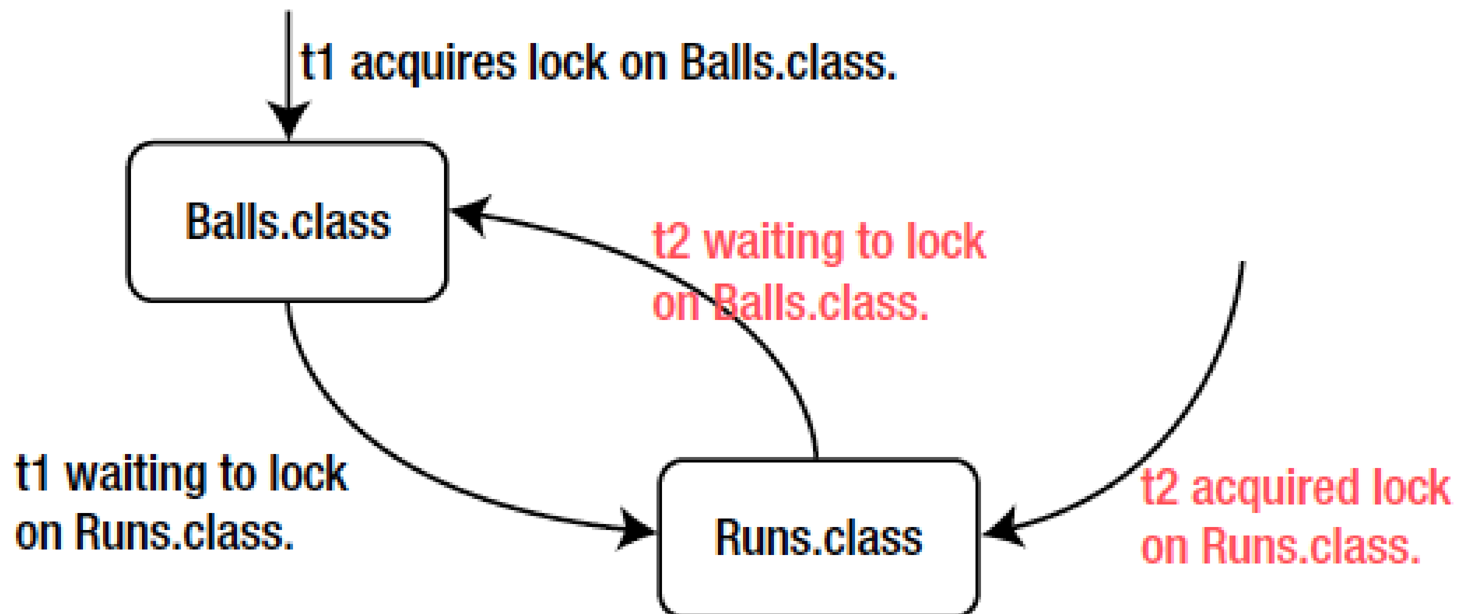
Deadlocks

- A deadlock arises when locking threads result in a situation where they cannot proceed and thus wait indefinitely for others to terminate.
- For some sequence of thread executions and the order in which locks are acquired and released, the program DeadLock.java in the previous slide may have a deadlock



Deadlock Prevention

- Deadlocks can arise in the context of multiple locks.
- If multiple locks are acquired in the same order, then a deadlock will not occur
- If multiple locks are acquired in a different order, then deadlocks may occur.



The Wait/Notify Mechanism

- The wait/notify mechanism provides a way for one thread to communicate to another thread.
- The method **wait()** allows the calling thread to wait for the wait object (on which wait() is called). A thread remains in the wait state until some another thread calls the **notify()** or **notifyAll()** method on the wait object.

CoffeeShop example

```java
class CoffeeMachine extends Thread {
    static String coffeeMade = null;
    static final Object lock = new Object();
    private static int coffeeNumber = 1;
    void makeCoffee() {
        synchronized(CoffeeMachine.lock) {
            if (coffeeMade != null) {
                try {
                    System.out.println("Coffee machine: Waiting for waiter notification to deliver the coffee");
                    CoffeeMachine.lock.wait();
                }
                catch (InterruptedException ie){
                    ie.printStackTrace();
                }
            }
            coffeeMade = "Coffee No. " + coffeeNumber++;
            System.out.println("Coffee machine says: Made " + coffeeMade);
            CoffeeMachine.lock.notifyAll();
            System.out.println("Coffee machine: Notifying waiter to pick the coffee ");
        }
    }

    public void run() {
        while(true) {
            makeCoffee();
            try {
                System.out.println("Coffee machine: Making another coffee now");
                Thread.sleep(3000);
            }
            catch(InterruptedException ie) {
                ie.printStackTrace();
            }
        }
    }
}
```

```java
class Waiter extends Thread {
    public void getCoffee() {
        synchronized(CoffeeMachine.lock) {
            if (CoffeeMachine.coffeeMade == null) {
                try {
                    System.out.println("Waiter: Will get orders till coffee machine notifies me ");
                    CoffeeMachine.lock.wait();
                }
                catch (InterruptedException ie) {
                    ie.printStackTrace();
                }
            }
            System.out.println("Waiter: Delivering " + CoffeeMachine.coffeeMade);
            CoffeeMachine.coffeeMade = null;
            CoffeeMachine.lock.notifyAll();
            System.out.println("Waiter: Notifying coffee machine to make another one");
        }
    }
    public void run() {
        while (true) {
            try {
               getCoffee();
               Thread.sleep(1000);
            }
            catch(InterruptedException ie) { }
        }
    }
}
```

```java
class CoffeeShop {
    public static void main(String[] args) {
        CoffeeMachine coffeeMachine = new CoffeeMachine();
        Waiter waiter = new Waiter();
        coffeeMachine.start();
        waiter.start();
    }
}
```

```
$ javac CoffeeMachine.java Waiter.java CoffeeShop.java
$ java CoffeeShop
```

```
Coffee machine says: Made Coffee No. 1
Coffee machine: Notifying waiter to pick the coffee
Coffee machine: Making another coffee now
Waiter: Delivering Coffee No. 1
Waiter: Notifying coffee machine to make another one
Waiter: Will get orders till coffee machine notifies me
Coffee machine says: Made Coffee No. 2
Coffee machine: Notifying waiter to pick the coffee
Coffee machine: Making another coffee now
Waiter: Delivering Coffee No. 2
Waiter: Notifying coffee machine to make another one
Waiter: Will get orders till coffee machine notifies me
Coffee machine says: Made Coffee No. 3
Coffee machine: Notifying waiter to pick the coffee
Coffee machine: Making another coffee now
Waiter: Delivering Coffee No. 3
Waiter: Notifying coffee machine to make another one
Waiter: Will get orders till coffee machine notifies me
Coffee machine says: Made Coffee No. 4
Coffee machine: Notifying waiter to pick the coffee
Coffee machine: Making another coffee now
Waiter: Delivering Coffee No. 4
Waiter: Notifying coffee machine to make another one
Waiter: Will get orders till coffee machine notifies me
Coffee machine says: Made Coffee No. 5
```

Exercises...

What are the results of the following programs?

```
class PingPong extends Thread {
        public void run() {
                System.out.println("ping ");
        }
        public static void main(String []args)  {
                Thread pingPong=new PingPong();
                pingPong.run();
                System.out.print("pong");
        }
}
```

```
class PingPong extends Thread {
        public void run() {
                System.out.println("ping");
        }
        public static void main(String []args)  {
                Thread pingPong=new PingPong();
                pingPong.start();
                System.out.println("pong");
        }
}
```

```
class PingPong extends Thread {
        public void run() {
                System.out.println("ping");
        }
        public static void main(String []args) throws InterruptedException{
                Thread pingPong=new PingPong();
                pingPong.start();
                pingPong.join();
                System.out.println("pong");
        }
}
```

What is the behavior of the following program?

```
class ExtendThread extends Thread {
        public void run() { System.out.print(Thread.currentThread().getName()); }
}

class ThreadTest{
        public static void main(String []args) throws InterruptedException {
                Thread thread1 = new Thread(new ExtendThread(), "thread1 ");
                Thread thread2 = new Thread(thread1, "thread2 ");
                thread1.start();
                thread2.start();
                thread1.start();        // START
        }
}
```

What is the behavior of the following program?

```
class ThreadTest {
    public static void main(String []args) throws InterruptedException {
        Thread t1=new Thread() {
            public void run() { System.out.print("t1 "); }
        };
        Thread t2=new Thread() {
            public void run() { System.out.print("t2 "); }
        };
        t1.start();
        t1.sleep(5000);
        t2.start();
        t2.sleep(5000);
        System.out.println("main ");
    }
}
```