

# Coordination and Scheduling Part 1



Concurrent and Parallel Programming

# Coordination of threads

---

- ▶ In some situations, there's the need to **coordinate the execution of the threads** of an application.
- ▶ For example, some threads might have to wait until one specific thread has completed its work.
- ▶ If a **thread need to wait** until some type of event happens, how can it be implemented in the source code?

# Example (1 / 3)

```
class Passenger extends Thread {  
    private Bus bus;  
  
    public Passenger(Bus bus) {  
        this.bus = bus;  
    }  
  
    @Override  
    public void run() {  
        System.out.println("passenger is waiting for the bus");  
  
        // NEED TO WAIT FOR THE BUS HERE!  
  
        System.out.println("passenger got notification");  
    }  
}
```

## Example (2/3)

```
class Bus extends Thread {
    private int numPassengers;

    public Bus(int numPassengers) {
        this.numPassengers = numPassengers;
    }

    @Override
    public void run() {
        try {
            sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("bus is arriving at the station");
        while (numPassengers > 0) {
            // NEED TO NOTIFY THE PASSENGERS HERE!
            numPassengers--;
            System.out.println("passenger is given notification call");
        }
    }
}
```

## Example (3 / 3)

```
class TestBus {  
    public static void main(String[] args) {  
        Bus bus = new Bus(3);  
        bus.start();  
        Passenger p1 = new Passenger(bus);  
        p1.start();  
        Passenger p2 = new Passenger(bus);  
        p2.start();  
        Passenger p3 = new Passenger(bus);  
        p3.start();  
  
        try {  
            bus.join();  
            p1.join();  
            p2.join();  
            p3.join();  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

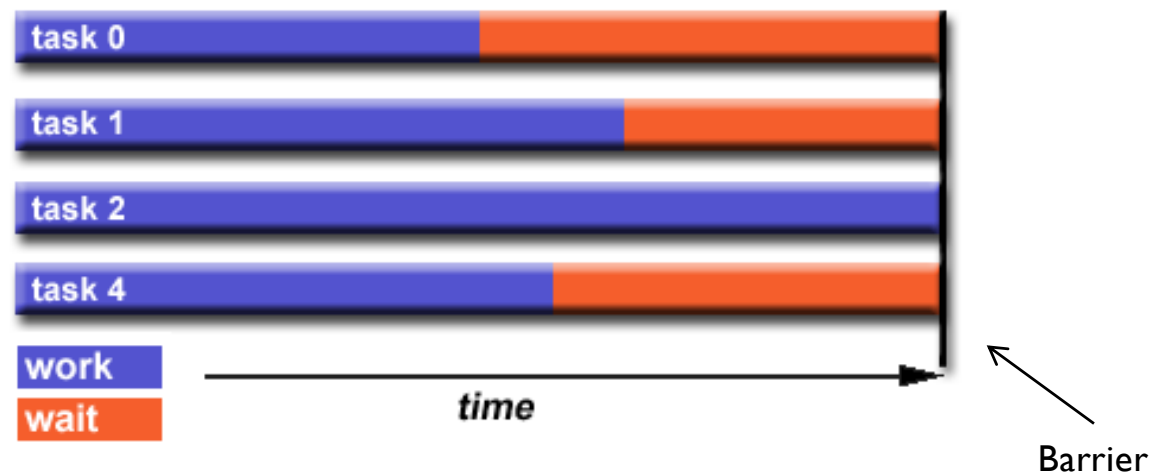
# Coordination of threads

---

- ▶ **Synchronization tools** such as synchronized blocks and ReentrantLocks prove useful when dealing with concurrent access to shared and mutable state.
- ▶ But, ... are these tools useful for the **explicit coordination of threads**?
- ▶ What are the alternatives? A sleep()? A while loop with a shared volatile flag?

# Coordination of threads

- ▶ To solve these problems, Java provides tools that are specifically designed to **coordinate the execution of the threads in an explicit way**.



## Wait(), notify() and notifyAll()

---

- ▶ **wait()** - the calling thread leaves the intrinsic lock that it has acquired (of the containing synchronized block or method) and goes to sleep until it is notified for wake up.
- ▶ **notify()** - wakes up the first thread that called **wait()** inside a synchronized block or method using the same intrinsic lock.
- ▶ **notifyAll()** wakes up all threads that called **wait()** inside a synchronized block or method using the same intrinsic lock. The thread with the highest priority starts first.



# Example (1 / 3)

```
class Passenger extends Thread {  
    private Bus bus;  
  
    public Passenger(Bus bus) {  
        this.bus = bus;  
    }  
  
    @Override  
    public void run() {  
        System.out.println("passenger is waiting for the bus");  
        synchronized (bus) {  
            try {  
                bus.wait();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
        System.out.println("passenger got notification");  
    }  
}
```

## Example (2/3)

```
class Bus extends Thread {  
    private int numPassengers;  
  
    public Bus(int numPassengers) {  
        this.numPassengers = numPassengers;  
    }  
  
    @Override  
    public void run() {  
        try {  
            sleep(1000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
  
        System.out.println("bus is arriving at the station");  
        while (numPassengers > 0) {  
            synchronized (this) {  
                notify();  
            }  
            numPassengers--;  
            System.out.println("passenger is given notification call");  
        }  
    }  
}
```

---

10 }

## Example (3 / 3)

---

passenger is waiting for the bus  
passenger is waiting for the bus  
passenger is waiting for the bus  
bus is arriving at the station  
passenger is given notification call  
passenger is given notification call  
passenger is given notification call  
passenger got notification  
passenger got notification  
passenger got notification

## Wait(), notify() and notifyAll()

---

- ▶ For reasons that will be clarify later, calls to wait(), notify() and notifyAll() **have to be performed in a synchronized block or method.**
- ▶ The intrinsic lock used for wait() **has to match with** the intrinsic lock used for notify() or notifyAll(). Otherwise an exception is thrown:

```
java.lang.IllegalMonitorStateException  
at java.lang.Object.wait(Native Method)  
at java.lang.Object.wait(Object.java:503)  
at Passenger.run(TestBus.java:13)
```

## Wait condition

- ▶ In the example that we analysed so far, the Passenger thread `waits()` in an unconditional way.
- ▶ However, `wait()` is usually performed conditionally:

```
if(!condition) {  
    wait();  
}
```

- ▶ But the following form has always to be used:

```
while(!condition) {  
    wait();  
}
```

## Example (1 / 2)

The while  
loop is  
mandatory!

```
public class BoundedBuffer {  
    private String[] buffer;  
    private int capacity;  
  
    private int front, rear, count;  
  
    public BoundedBuffer(int capacity) {  
        this.capacity = capacity;  
        buffer = new String[capacity];  
    }  
  
    public synchronized void deposit(String data) {  
        while (count == capacity) {  
            try {  
                wait();  
            } catch (InterruptedException e) {}  
        }  
  
        buffer[rear] = data;  
        rear = (rear + 1) % capacity;  
        count++;  
  
        notifyAll();  
    }  
}
```

## Example (2/2)

The while  
loop is  
mandatory!

```
// ...

public synchronized String fetch() {
    ➤ while (count == 0) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }

    String result = buffer[front];
    front = (front + 1) % capacity;
    count--;

    notifyAll();

    return result;
}
}
```

# Coordination of threads

- ▶ The while loop is mandatory, because it happens that some threads wake up at the wrong moment.
- ▶ Therefore wait() has always to be called in a **loop that checks the wait condition again.**

## Spurious wake-ups - reasons to check the condition again:

- ▶ Threads wake up but don't start running straightaway. They need to be rescheduled. In the meantime, the execution condition could have changed once again.
- ▶ notify() wakes up the first thread waiting, but is it the right one?
- ▶ notifyAll() wakes up everyone, but do they all have to start?



## But how exactly does it work?

---

- ▶ As soon as the thread calls `wait()`, **the lock is released**. This allows other threads to acquire the lock, instead of getting blocked by the waiting thread.
- ▶ Subsequently, when another thread calls `notify()`, it wakes up the waiting thread.
- ▶ When the thread is again active, **it needs to reacquire the lock** before being able to execute the code after the `wait()`. It must therefore wait, **at least until the thread that made the call to `notify()` exits the lock-protected code region**.

## But why?

- ▶ But why is it necessary to call `wait()` and `notify()` inside a lock?
  - ▶ `wait()` and `notify()` must coordinate on a common object. Use of the intrinsic lock, allows to have more than one independent wait/notify coordination point within the program.
  - ▶ `wait()` and `notify()` are tools for communication between threads. Use of `synchronized` is of help for correct memory management.
  - ▶ The `wait()` condition, which needs to be checked before going to sleep, is a shared and mutable state modified by the thread that performs the wake-up. As a consequence, the **condition check and the `wait()` are a compound action that needs to be atomic!**

# Example

```
synchronized (obj) {  
    while (<condition does not hold>)  
        obj.wait(timeout);  
    // Perform action appropriate to condition  
}
```

```
synchronized (obj) {  
    <modify condition>  
    obj.notify();  
}
```

# Wait() and race-conditions

---

## Be careful:

- ▶ The wait() operation forces the thread to release the lock before going to sleep.
- ▶ Release of the lock is mandatory **to avoid deadlock situations**: the other threads must be able to enter a synchronized block/method to call notify().
- ▶ However, be **very careful about potential race-condition conditions!** The thread leaves the lock at the wait() operation, ... therefore, there is an additional window of vulnerability.

```
synchronized (this) {  
    if (obj == null) {  
        while (!notReady) {  
            try {  
                wait();  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
        obj = new SharedObject();  
    }  
}
```

- ▶ Use of `wait()` and `notify()`, in combination with `synchronized blocks/methods`, is not always an ideal solution to coordinate the execution of the threads. For example, it is not allowed to have more than one independent wait/notify coordination point for the same intrinsic lock.
- ▶ As an alternative, `conditions` (in combination with explicit locks) can be used. The `java.util.concurrent.locks` package provides the `Condition` class: support similar to "wait" and "notify" but more flexible.

► **java.util.concurrent.locks**

Interface Summary	
Interface	Description
<a href="#">Condition</a>	Condition factors out the <code>Object</code> monitor methods ( <a href="#">wait</a> , <a href="#">notify</a> and <a href="#">notifyAll</a> ) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary <a href="#">Lock</a> implementations.
<a href="#">Lock</a>	<code>Lock</code> implementations provide more extensive locking operations than can be obtained using <code>synchronized</code> methods and statements.
<a href="#">ReadWriteLock</a>	A <code>ReadWriteLock</code> maintains a pair of associated <a href="#">locks</a> , one for read-only operations and one for writing.

Class Summary	
Class	Description
<a href="#">AbstractOwnableSynchronizer</a>	A synchronizer that may be exclusively owned by a thread.
<a href="#">AbstractQueuedLongSynchronizer</a>	A version of <a href="#">AbstractQueuedSynchronizer</a> in which synchronization state is maintained as a <code>long</code> .
<a href="#">AbstractQueuedSynchronizer</a>	Provides a framework for implementing blocking locks and related synchronizers (semaphores, events, etc) that rely on first-in-first-out (FIFO) wait queues.
<a href="#">LockSupport</a>	Basic thread blocking primitives for creating locks and other synchronization classes.
<a href="#">ReentrantLock</a>	A reentrant mutual exclusion <a href="#">Lock</a> with the same basic behavior and semantics as the implicit monitor lock accessed using <code>synchronized</code> methods and statements, but with extended capabilities.
<a href="#">ReentrantReadWriteLock</a>	An implementation of <a href="#">ReadWriteLock</a> supporting similar semantics to <a href="#">ReentrantLock</a> .
<a href="#">ReentrantReadWriteLock.ReadLock</a>	The lock returned by method <a href="#">ReentrantReadWriteLock.readLock()</a> .
<a href="#">ReentrantReadWriteLock.WriteLock</a>	The lock returned by method <a href="#">ReentrantReadWriteLock.writeLock()</a> .

# Example (1 / 3)

```
public class BoundedBuffer {  
    private final String[] buffer;  
    private final int capacity;  
  
    private int front;  
    private int rear;  
    private int count;  
  
    private final Lock lock = new ReentrantLock();  
    private final Condition notFull = lock.newCondition();  
    private final Condition notEmpty = lock.newCondition();  
  
    public BoundedBuffer(final int capacity) {  
        super();  
        this.capacity = capacity;  
        buffer = new String[capacity];  
    }  
}
```



## Example (2/3)

```
public void deposit(final String data) throws InterruptedException {  
    lock.lock();  
    try {  
        while (count == capacity) {  
            notFull.await();  
        }  
  
        buffer[rear] = data;  
        rear = (rear + 1) % capacity;  
        count++;  
  
        notEmpty.signal();  
    } finally {  
        lock.unlock();  
    }  
}
```

## Example (3 / 3)

```
public String fetch() throws InterruptedException {  
    lock.lock();  
    try {  
        while (count == 0) {  
            notEmpty.await();  
        }  
  
        final String result = buffer[front];  
        front = (front + 1) % capacity;  
        count--;  
  
        notFull.signal();  
  
        return result;  
    } finally {  
        lock.unlock();  
    }  
}
```

- ▶ When conditions are used, the **same programming rules for wait and notify** must be respected:
  - ▶ The check and the call to `await()`, as well as the call to `signal()`, have always to be performed inside a code region protected by an explicit lock.
  - ▶ The same lock has to be used for the `await()`, as well as for the `signal()` parts.
  - ▶ The check has to be performed in a while loop.

# Synchronizers

- ▶ **Synchronizers** are Concurrent Building Blocks, useful for coordinating the flow of execution of concurrently executing threads.
- ▶ Synchronizers are implemented using Conditions.
- ▶ General behaviour:
  - ▶ The thread asks the synchronizer to check the internal state
  - ▶ According to the internal state the synchronizer forces the thread to wait or let it pass
  - ▶ The synchronizer provides methods for other threads to manipulate the state

Queues are part of the synchronizers. The main variants are:

- ▶ *ConcurrentLinkedQueue (FIFO) and PriorityQueue*: contain a set of items to be processed. Don't have blocking behaviour. If the queue is empty, null is returned.
- ▶ **Blocking queues**: blocks when the queue is empty (in case of element removal) and when the queue is full (in case of element addition).

Blocking queues are useful to implement the **producer-consumer design pattern** (used by the task executor framework) - will be discussed in the next lessons.

# ConcurrentLinkedQueue (1/2)

```
class Getter implements Runnable {  
    private final Queue<MyState> queue;  
    private final int id;  
  
    public Getter(Queue<MyState> queue, int id) {  
        this.queue = queue;  
        this.id = id;  
    }  
  
    @Override  
    public void run() {  
        while (ConcurrentLinkedQueueExample.isRunning) {  
            MyState state = queue.poll();  
            if (state != null)  
                System.out.println("Getter[" + id + "] got " + state);  
        }  
    }  
}
```

```
class MyState {  
    private int id;  
  
    public MyState(int id) {  
        this.id = id;  
    }  
  
    public String toString() {  
        return "MyState" + id;  
    }  
}
```

# ConcurrentLinkedQueue (2/2)

```
public class ConcurrentLinkedQueueExample {
    static volatile boolean isRunning = true;

    public static void main(String[] args) throws InterruptedException {
        final Queue<MyState> queue = new ConcurrentLinkedQueue<>();
        final List<Thread> allThreads = new ArrayList<>();
        for (int i = 1; i <= 5; i++)
            allThreads.add(new Thread(new Getter(queue, i)));
        allThreads.forEach(Thread::start);

        for (int i = 0; i < 1000; i++) {
            queue.add(new MyState(i));
            Thread.sleep(1);
        }
        isRunning = false;

        for (Thread thread : allThreads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Terminated");
    }
}
```

```
...
Updater[2] got MyState284
Updater[3] got MyState285
Updater[3] got MyState286
Updater[5] got MyState287
Updater[1] got MyState288
Updater[2] got MyState289
...
```

## Queues' methods:

Method Behaviour	Insert	Remove	Examine
Throw exception	<code>add(e)</code>	<code>remove()</code>	<code>element()</code>
Returns value	<code>offer(e)</code>	<code>poll()</code>	<code>peek()</code>
Time out	<code>offer(e, time, timeunit)</code>	<code>poll(time, timeunit)</code>	-
<b>Block*</b>	<code>put(e)</code>	<code>take()</code>	-



# Types of BlockingQueues

## Types of BlockingQueues:

Queue	Data Structure	Capacity	Insertion Policy
LinkedBlockingQueue	Linked nodes	Optionally bounded	FIFO
ArrayBlockingQueue	Fixed size array	Bounded	FIFO
PriorityBlockingQueue	-	Unbounded	Comparable
DelayQueue	-	Unbounded	Delayed elements

- ▶ The *SynchronousQueue* is also a very simple form of Synchronizer. It is used by two threads to exchange a value.
- ▶ At the practical level, it is a queue without capacity. It synchronizes immediately as soon as both threads arrives at the synchronization point. As long as only one thread is there, it has to wait.
- ▶ The *Deque* functionality is of help for insertion and removal at both sides of the queue.

# Example SynchronousQueue (1/2)

```
class Data {
    private double radius = 1.0;
    private double area = 2.0 * Math.PI * radius * radius;

    public void update(double step) {
        radius *= step;
        area = 2.0 * Math.PI * radius * radius;
    }

    public String toString() {
        return "radius=" + radius + " area=" + area;
    }
}

class Worker implements Runnable {
    public void run() {
        while (true)
            try {
                Data data = SynchronousQueueExample.queue.take();
                System.out.println("Worker: " + data);
                data.update(2.1);
                SynchronousQueueExample.queue.put(data);
            } catch (InterruptedException e) {
                System.out.println("Worker interrupted. Terminating");
                return;
            }
    }
}
```

# Example SynchronousQueue (2/2)

```
public class SynchronousQueueExample {
    static final SynchronousQueue<Data> queue = new SynchronousQueue<>();

    public static void main(String[] args) throws InterruptedException {
        final Thread worker = new Thread(new Worker());
        worker.start();
        queue.put(new Data());
        Thread generator = new Thread(() -> {
            for (int i = 0; i < 10; i++)
                try {
                    Data sharedObject = queue.take();
                    System.out.println("Main : " + sharedObject);
                    sharedObject.update(0.5);
                    queue.put(sharedObject);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            worker.interrupt();
        });
        generator.start();
        try {
            generator.join();
            worker.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
Worker: radius=1.0 area=6.283185307179586
Main : radius=2.1 area=13.194689145077131
Worker: radius=1.05 area=6.5973445725385655
Main : radius=2.205 area=13.854423602330987
Worker: radius=1.1025 area=6.927211801165494
Main : radius=2.31525000000003 area=14.547144782447539
Worker: radius=1.15762500000001 area=7.2735723912237695
Main : radius=2.43101250000005 area=15.274502021569917
Worker: radius=1.21550625000002 area=7.637251010784959
Main : radius=2.552563125000007 area=16.038227122648415
Worker: radius=1.2762815625000004 area=8.019113561324207
Main : radius=2.680191281250001 area=16.840138478780837
Worker: radius=1.3400956406250004 area=8.420069239390418
Main : radius=2.814200845312501 area=17.68214540271988
Worker: radius=1.4071004226562505 area=8.84107270135994
Main : radius=2.954910887578126 area=18.56625267285587
Worker: radius=1.477455443789063 area=9.283126336427935
Main : radius=3.1026564319570324 area=19.494565306498664
Worker: radius=1.5513282159785162 area=9.747282653249332
Main : radius=3.257789253554884 area=20.4692935718236
Worker: radius=1.628894626777442 area=10.2346467859118
Worker interrupted. Terminating
```

# Semaphore

- ▶ A frequently used synchronizer is the *Semaphore*: controls the access to resources (for example a pool of objects).
- ▶ The Semaphore *releases n virtual access permissions* via the `acquire()` and `release()` methods.
- ▶ A *binary semaphore*, which releases only one permission, is similar to a mutex.

## Example (1 / 2)

```
class Pool {  
    private static final int MAX_AVAILABLE = 100;  
    private final Semaphore available = new Semaphore(MAX_AVAILABLE, true);  
  
    public Object getItem() throws InterruptedException {  
        available.acquire();  
        return getNextAvailableItem();  
    }  
  
    public void putItem(final Object x) {  
        if (markAsUnused(x))  
            available.release();  
    }  
  
    protected Object[] items; // = ... whatever kinds of items being managed  
    protected boolean[] used = new boolean[MAX_AVAILABLE];  
}
```

## Example (2/2)

```
protected synchronized Object getNextAvailableItem() {
    for (int i = 0; i < MAX_AVAILABLE; ++i) {
        if (!used[i]) {
            used[i] = true;
            return items[i];
        }
    }
    return null; // not reached
}

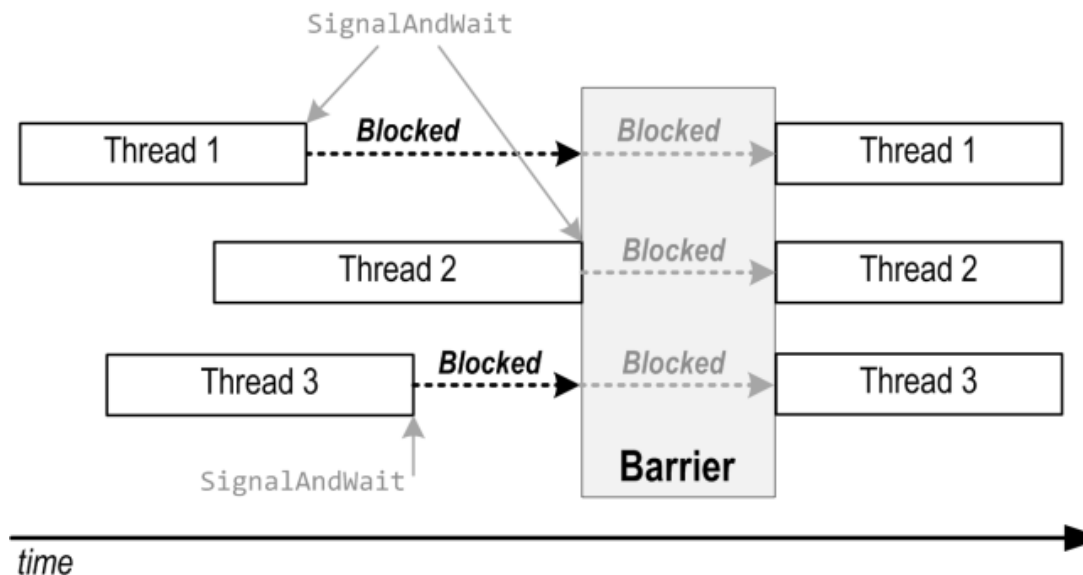
protected synchronized boolean markAsUnused(final Object item) {
    for (int i = 0; i < MAX_AVAILABLE; ++i) {
        if (item == items[i]) {
            if (used[i]) {
                used[i] = false;
                return true;
            } else
                return false;
        }
    }
    return false;
}
```

Other available Synchronizers are:

- ▶ ***CountDownLatch***: threads have to wait until the latch reaches its terminal state. Similar to a door.  
Example of use: to simultaneously start the execution of threads.
- ▶ ***CyclicBarrier***: similar to the CountDownLatch, but it is possible to reset it for reuse.
- ▶ ***Exchanger***: allows threads to exchange objects at a rendez-vous. Blocks until another thread calls exchange(). Useful for sharing data buffers. Similar behaviour of the *SynchronousQueue*.



- **Phaser**: similar functionality to `CountDownLatch` and `CyclicBarrier`, but more flexible. If there's the need to wait that a thread completes some work before continuing with other operations, the phaser is an ideal solution.



## Example

```

class Player implements Runnable {
    private final int id;

    public Player(final int id) {
        this.id = id;
        PhaserExample.rouletteTable.register();
    }


    @Override
    public void run() {
        System.out.println("Player" + id + " joining table");
        int wins = 0, loss = 0;
        while (loss < 10 && wins < 10) {
            // Place bet and wait for the croupier to spin the wheel
            int bet = ThreadLocalRandom.current().nextInt(0, 37);
            int game = PhaserExample.rouletteTable.arriveAndAwaitAdvance();
            int win = PhaserExample.winningNumber;
            System.out.println("Player" + id + " game=" + game + " bet=" + bet
                               + " winningNumber=" + win);

            if (bet == win)
                wins++;
            else
                loss++;

            try {
                Thread.sleep(ThreadLocalRandom.current().nextLong(10, 50));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // Leave table
        int game = PhaserExample.rouletteTable.arriveAndDeregister();
        System.out.println("Player" + id + " leaving. game=" + game
                           + " wins=" + wins + " loss=" + loss);
    }
}

```

Perform read after  
arriveAndAwaitAdvance()



## Example

```

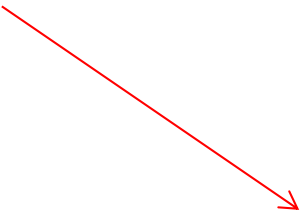
public class PhaserExample {
    final static Phaser rouletteTable = new Phaser(1);
    static int winningNumber = -1;

    public static void main(String[] args) throws InterruptedException {
        final List<Thread> allThreads = new ArrayList<>();
        int id = 1;
        while (true) {
            // Add more players (limit to 30 max)
            int newPlayers = id > 30 ? 0 : ThreadLocalRandom.current().nextInt(1, 3);
            for (int j = 0; j < newPlayers; j++) {
                final Thread t = new Thread(new Player(id++));
                allThreads.add(t);
                t.start();
            }
            // Let players place their bets
            Thread.sleep(100);
            // main is a registered party, remove it from players count!
            int numberOfPlayers = rouletteTable.getRegisteredParties() - 1;
            if (numberOfPlayers == 0)
                break;
            int winNumber = ThreadLocalRandom.current().nextInt(37);
            System.out.println("Rien ne va plus. Players=" + numberOfPlayers
                               + " Winning number=" + winNumber);
            // simulates the croupier spinning the wheel
            winningNumber = winNumber;
            rouletteTable.arrive();
        }
        for (Thread thread : allThreads)
            thread.join();
    }
}

```

Perform write before arrive()

arriveAndAwaitAdvance() and  
arrive() raise the memory  
barrier!



# Example

Player1 joining table

Player2 joining table

Rien ne va plus. Number of players: 2 Winning number=1

Player2: game=1 bet=2 winningNumber=1

Player1: game=1 bet=4 winningNumber=1

Player3 joining table

Player4 joining table

Rien ne va plus. Number of players: 4 Winning number=21

Player2: game=2 bet=31 winningNumber=21

Player4: game=2 bet=22 winningNumber=21

Player1: game=2 bet=1 winningNumber=21

Player3: game=2 bet=3 winningNumber=21

Player5 joining table

Player6 joining table

# Synchronizers and visibility

---

- ▶ Synchronizers like the `CyclicBarrier` **guarantee correct memory visibility**.
- ▶ The Javadoc provides information about this guarantee: "memory consistency effects: actions performed by a thread before the call to `await()` happen-before actions that are part of the memory barrier, which in turn happen before actions that follow a correct return by the corresponding `await()` in other threads".
- ▶ In other words, the `await()` method **includes a call to the memory barrier**.