Java Monitors

Monitor

- A thread synchronization mechanism for ensuring exclusive access to a data object that has a certain condition or state
- If the condition is not met, a thread gives up exclusive access and waits until another thread sets the appropriate condition on the data object.
- Monitors can be created in Java using synchronized blocks and methods from java.lang.Object

Java Monitors

- synchronized block guarantees exclusive access to a critical section
- java.lang.Object.wait() - causes a thread to give up exclusive access and block until another thread unblocks it.
  - Called when required condition is not met
- java.lang.Object.notifyAll() - unblocks all threads waiting on this object's monitor

Producer-Consumer Problem

- A classic thread synchronization problem
- A producer thread creates items and puts them into a fixed-size queue. The producer requires that there is at least one empty space in the queue. If the queue is full, the producer must wait.
- A consumer thread removes items from the queue. The consumer requires that there is at least one item in the queue. If the queue is empty, the consumer must wait.
- Producer and consumer both require exclusive access to the queue

Producer

- The producer has exclusive access to the queue inside the synchronized block.
- While the required producer’s condition is not met, the producer calls the queue object’s wait().
    - This allows the producer give up exclusive access to the queue while remaining inside its synchronized block.
    - Another thread can get exclusive access to the queue.
    - The producer will unblock when another thread calls notifyAll(). The producer will not get regain exclusive access until the other thread exits its synchronized block.
        - Upon regaining exclusive access, the producer retests the condition.
- If the condition of the queue is met, the producer adds an item to the queue, calls notifyAll() to unblock any waiting thread, and exits its critical section (its synchronized block).

Producer Example

```java
public class Producer implements Runnable {

    private final List<Integer> queue;
    private final int maxCapacity;
    private static int counter = 0;

    public Producer(List<Integer> sharedQueue, int size) {
        queue = sharedQueue;
        maxCapacity = size;
    }

    @Override
    public void run() {
        while (true) {
            try {
                produce(counter++);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

     // see next column
}
```

```java
private void produce(int i) throws InterruptedException {
String name = Thread.currentThread().getName();
synchronized(queue) {
    while (queue.size() == maxCapacity) {
        System.out.println("Queue is full. "
                + name + " is waiting. Size: " + queue.size());
        taskQueue.wait();
    }

    Thread.sleep(100);
    queue.add(i);
    System.out.println(name + " produced: " + i);
    queue.notifyAll();
}
}
```

Consumer

- The consumer has exclusive access to the queue inside the synchronized block.
- While the consumer’s required condition is not met, the consumer calls queue object’s wait().
  - This allows the consumer give up exclusive access to the queue while remaining inside its synchronized block.
  - Another thread can get exclusive access to the queue.
  - The consumer will unblock when another thread calls notifyAll(). The producer will not get regain exclusive access until the other thread exits its synchronized block.
    - Upon regaining exclusive access, the consumer retests the condition.
- If the condition of the queue is met, the consumer adds an item to the queue, calls notifyAll() to unblock any waiting thread, and exits its critical section (its synchronized block)

Consumer Example

```
public class Consumer implements Runnable {

	private final List<Integer> queue;

	public Consumer(List<Integer> sharedQueue) {
		queue = sharedQueue;
	}

	@Override
	public void run() {
		while (true) {
			try {
				consume();
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
		}
	}

	// see next column
}
```

```
private void consume() throws InterruptedException {
	String name = Thread.currentThread().getName();
	synchronized(queue) {
		while (queue.isEmpty()) {
			System.out.println("Queue is empty. "
					+ name + " is waiting. Size: " + queue.size());
			queue.wait();
		}
		Thread.sleep(100);
		int i = (Integer) queue.remove(0);
		System.out.println(name + " consumed: " + i);
		queue.notifyAll();
	}
}
```

Final

- This example requires at least one producer and one consumer.
- Supports multiple producers and consumers.
- What will happen if the programmer forgets to call queue.wait() or queue.notifyAll()?
- Why is Producer.counter declared static?