Thread synchronization and communication

While threads are often used to work on a task more or less independently from other threads, there are many occasions where one would want to pass data between threads, or even control other threads, such as from a central task scheduler thread.

The central problem with concurrency is that of ensuring safe access to shared resources, including when communicating between threads. There is also the issue of threads being able to communicate and synchronise themselves.

What makes **multithreaded programming** such a challenge is to be able to keep track of each interaction between threads, ensure that each and every form of access is secured while not falling into the traps of **deadlocks** and **race conditions**.

Shared memory

Often, threads need to communicate with each other. They do this by sharing an object. Let's develop an **example** in which several threads share an object of class Counter.

In the example, some threads cooperate with each other to increment a counter. The counter is implemented as an object of a Counter class that is passed to each thread in its constructor. Therefore, all threads share this object. They can all query its value, and they can all modify it. The main thread creates and launches all threads and waits for them to finish their execution using the join method.

The Counter class can be implemented as follows:

```
public class Counter {
    private int count=0;
    public int getCount() {
        return count;
    }
    public int increase(){
        this.count++;
        return count;
    }
}
```

The code of each thread could be:

```
public class CounterThread implements Runnable{
    private Counter counter;
    private static final int NUM_OPS=300;
    public CounterThread(Counter counter) {
        this.counter = counter;
    }
    @Override
    public void run() {
        for (int i = 0; i < NUM_OPS; i++) {
            try {
                counter.increase();
                System.out.println(Thread.currentThread().getName() + " " +
counter.getCount());
```

```
                Thread.sleep((long) (Math.random() * 100 + 50));
            } catch (InterruptedException ex) {
                throw new RuntimeException(ex);
            }
        }
    }
}
```

The program that launches the threads would be:

```
public class LaunchCounterThread {
    private static final int NUM_THREADS=10;
    public static void main(String[] args) throws InterruptedException {

        //Initialize the counter object
        Counter counter = new Counter();

        // Create ten threads
        Thread[] th=new Thread[NUM_THREADS];
        for (int i = 0; i < NUM_THREADS; i++) {
            CounterThread ct=new CounterThread(counter);
            th[i]=new Thread(ct,"CounterThread"+i);
            th[i].start();
        }
        for (int i = 0; i < NUM_THREADS; i++) {
            th[i].join();
        }
        System.out.println("The final value of counter is " + counter.getCount());
    }
}
```

With the above program, the counter does not reach the value you would expect. This is because if two threads read a value simultaneously, increment it, and then write it back, one increment could be lost. This is very commonly also known as a **race condition**.

Atomic operations

Atomicity is about performing atomic operations that complete in a single step relative to other threads. When we talk about atomic variables and atomic operations, we refer to indivisible actions. This means that no other thread can see the operation in an incomplete state. In Java, atomicity is often achieved using atomic classes provided by the **java.util.concurrent.atomic** package, which supports atomic operations on single variables.

Atomic operations are crucial in scenarios where simple **read-modify-write operations**, such as incrementing a counter, must be thread-safe without using synchronization mechanisms like the synchronized keyword or synchronized blocks, which can lead to thread contention and affect performance.

```
public class SafeCounter {
    private AtomicInteger count=new AtomicInteger(0);
    public int getCount() {return count.get();}
    public int increase(){return count.incrementAndGet();}
}
```

In the class **SafeCounter,** count is an **atomic variable**. The incrementAndGet() method, which is an atomic operation, increments the count by one and returns the updated value as an atomic step. This guarantees that even when multiple threads are invoking increment() simultaneously, each call to **incrementAndGet()** is executed as a single, indivisible operation, ensuring thread safety without the overhead of locking.

Atomic operations are critical in concurrent programming, particularly in environments where multiple threads manipulate shared data. Understanding how these operations work and why they are beneficial can significantly impact the performance and reliability of multi-threaded applications.

Some of the **classes** in **java.util.concurrent.atomic** package are:

- **AtomicInteger**. Provides atomic operations for integers.
- **AtomicLong**. Provides atomic operations for long integers.
- **AtomicBoolean**. Privides atomic operations for booleans.
- **AtomicReference**. Provides atomic operations for object references.

These classes provide methods to perform common operations such as incrementing, setting, comparing, and updating values in an atomic, thread-safe manner.

Common **methods** in **atomic classes** are shown in the following table.

Method	Description
void get()	Returns the current value.
set(value)	Sets the value to a new value.
getAndSet(newValue)	Atomically sets the value to a new value and returns the old value.
compareAndSet(expectedValue,newValue)	Compares the current value with an expected value and, if they are equal, set it to a new value.
incrementAndGet() / getAndIncrement()	Increments the value atomically.
decrementAndGet() / getAndDecrement()	Decrements the value atomically.
addAndGet(delta / getAndAdd(delta)	Adds a value atomically.

The above code does not work because the operations performed in the increment method are not atomic but are decomposed into simpler operations that are executed one after the other.

When these operations are executed in a thread, the execution of the thread can be interrupted, and operations of other threads can be interleaved between them. Depending on how the operations are interleaved and the data they access, unexpected results can be obtained. This is known as **race conditions**.

Communication between threads occurs mainly through shared access to objects and their properties. This communication mechanism is very efficient but presents two types of errors:

- Interference between threads
- Consistency errors of the information in memory

The programming tool used to prevent this type of error is **synchronization**.

Most of the time, threads don't care about the other threads running in the program or what they do. But if they need something from another thread, then they need synchronization.

The operations to increment the counter value should be executed without interleaving the same operations of another thread. They are said to constitute a **critical section**. To avoid **race conditions**, critical sections of different threads should be executed in **mutual exclusion**. That is, the critical section should not be running concurrently on more than one thread.

In Java, the reserved word **synchronized** is used to get **mutual exclusion**. To execute a block or a synchronized method, the threads must first get the lock of the object, waiting for it to be released if it has been adquired by another thread.

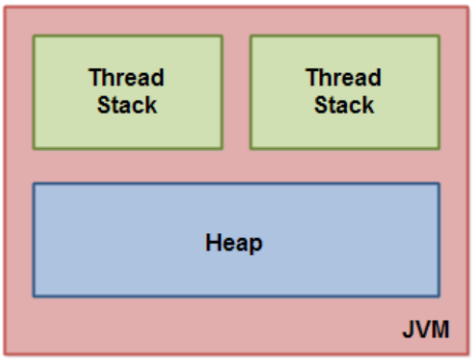
A class may have more than one synchronized method. A **synchronized** method will never be executed for an object if any synchronized method is already being executed on any other thread.

You can verify that, by simply adding **synchronized** to the beginning of the method declaration **increase**, the count always reaches the expected final value.

The **getCount** method is also declared as **synchronised** so that the counter value is not queried while it is being modified in another thread. With these changes, mutual exclusion is guaranteed in the access to the methods of the counter class.

```
public class Counter {
    private int count=0;
    synchronized public int getCount() {
        return count;
    }

    synchronized public int increase(){
        this.count++;
        return count;
    }
}
```

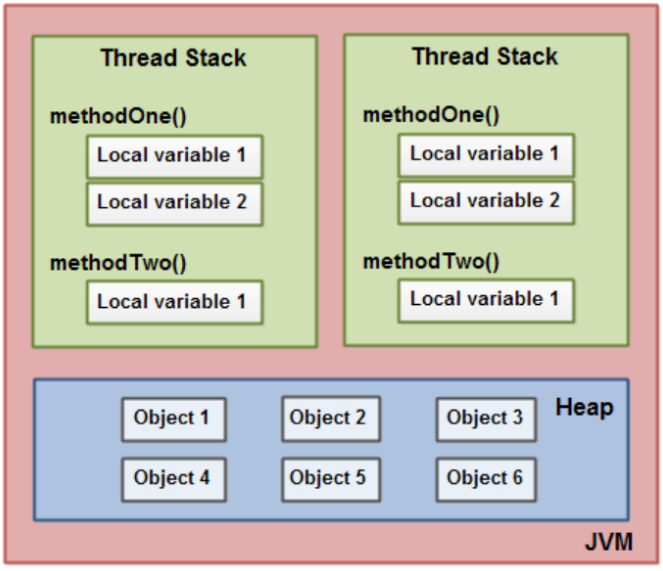
The Java **memory model** used internally in the JVM divides memory between thread stacks and the heap. This diagram illustrates the Java memory model from a logic perspective:



Each thread running in the JVM has its own thread stack. The thread stack contains information about what methods the thread has called to reach the current point of execution. As the thread executes its code, the call stack changes.

The **heap** contains all objects created in a Java application, regardless of what thread created the object. It does not matter if an object was created and assigned to a local variable, or created as a member variable of another object, the object is still stored on the heap.

Here is a diagram illustrating the call stack and local variables stored on the thread stacks, and objects stored on the heap:



Objects on the heap can be accessed by all threads that have a reference to the object.

Intrinsic lock

The implementation of the synchronised keyword is based on the **intrinsic lock** mechanism.

Every Java object has an **intrinsic lock** just because it is an **object**, that is, because it belongs to the Object class.

Nothing prevents a thread from reacquiring an intrinsic lock that it has already acquired. This is called **reentrant synchronisation**.

As long as a thread owns an intrinsic 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 until the lock is released by the currently executing thread.

The **synchronised** keyword can be applied in different types of Java code blocks, and in each case, a different locking object will be used.

- **Non-static methods**. Synchronized is added to the method declaration. Intrinsic locking is performed on the object on which the method is executed (this). Therefore, two different threads cannot both be running a non-static synchronised method on the same object at the same time.
- **Static methods**. Intrinsic lock is performed on the class to which the method belongs. Therefore, two different threads cannot both be executing a synchronised static method of the same class at the same time.
- **Synchronized statements**. Another way to create synchronized code is with synchronized statements. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```
synchronized(object) {
  (…)
}
```

A thread cannot execute a block of synchronised code on an object if there is already a thread that has acquired the lock for that object. The object should be of type **final**, because if a new value is assigned to it, all existing locks on that object are invalidated.

If the execution of a thread reaches a language element in which the **synchronised** keyword is present, and the intrinsic lock for the corresponding object is free, the thread is granted the lock. Once the code of the language element is **executed**, the lock is **released**. If the thread already holds the lock, it may simply re-execute the language element code (reentrant synchronization).

Below, we see the code of the previous **Counter** class renamed to **CounterBlock** with synchronised blocks. This allows a finer control of the synchronisation.

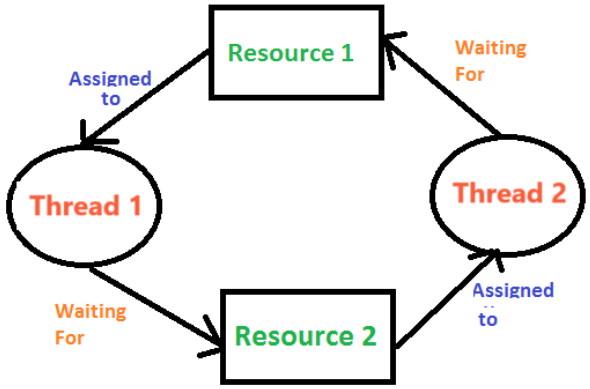
```
public class CounterBlock {
    private int count=0;
    private final Object lock=new Object();

    public int getCount() {
        synchronized(lock){
            return count;
        }
    }
```

```
    public int increase(){
        synchronized(lock) {
            this.count++;
            return count;
        }
    }
}
```

Resource sharing. Deadlock

In a multiprogramming environment, more than one process may compete for a finite set of resources. If a process requests for a resource and the resource is not presently available, then the process waits for it. Sometimes this waiting process never succeeds to get access to the resource. This waiting for resources leads to three scenarios: **deadlock**, **livelock**, and **starvation**.

A **deadlock** can occur when two threads become mutually locked, each waiting for the other to unlock an object it has locked.



The following code is an example of a deadlock in Java:

```
public class CreateDeadlock {
    public static void main(String[] args) {
        final String resource1 = "Resource 1";
        final String resource2 = "Resource 2";
        // Thread 1 tries to lock Resource 1 and then Resource 2
        Thread t1 = new Thread() {
            public void run() {
                synchronized (resource1) {
                    System.out.println("Thread 1: I have locked Resource 1");
                    try { Thread.sleep(50);} catch (Exception e) {}
                    synchronized (resource2) {
                        System.out.println("Thread 1: I have locked Resource 2");  }
                }
            }
        };
// Thread 2 tries to lock Resource 2 then Resource 1
        Thread t2 = new Thread() {
            public void run() {
                synchronized (resource2) {
                    System.out.println("Thread 2: I have locked Resource 2 ");
                    try { Thread.sleep(50);} catch (Exception e) {}
                    synchronized (resource1) {
                        System.out.println("Thread 2: I have locked Resource 2");
```

```
                    }
                }
            }
        };
        t1.start();
        t2.start();
    }
}
```

To resolve this deadlock, we can simply change the order of acquiring the locks.

So, one way to prevent deadlock is to put an ordering on the locks that need to be acquired simultaneously, and ensuring that all code acquires the locks in that order.

```
public class ResolveDeadlock {
    public static void main(String[] args) {
        final String resource1 = "Resource 1";
        final String resource2 = "Resource 2";
        // Thread 1 tries to lock Resource 2 and then Resource 1
        Thread t1 = new Thread() {
            public void run() {
                synchronized (resource2) {
                    System.out.println("Thread 1: I have locked Resource 2");
                    try { Thread.sleep(50);} catch (Exception e) {}
                    System.out.println("Thread 1: I am waiting for Resource 1");
                    synchronized (resource1) {
                        System.out.println("Thread 1: I have locked Resource 1 and 2");
                    }
                }
            }
        };
// Thread 2 tries to lock Resource 2 and then Resource 1
        Thread t2 = new Thread() {
            public void run() {
                synchronized (resource2) {
                    System.out.println("Thread 2: I have locked Resource 2 ");
                    try { Thread.sleep(50);} catch (Exception e) {}
                    System.out.println("Thread 2: I am waiting for Resource 1");
                    synchronized (resource1) {
                        System.out.println("Thread 2: I have locked Resource 1");
                    }
                }
            }
        };
        t1.start();
        t2.start();
    }
}
```

Oracle says that you can use any object as a **sync monitor**, however they recommend not to sync on String, or any wrapper object of primitive data types (Integer, Double, Boolean, ...).

To be on the safe side, it is best to synchronise on this, on an instance of an object or, failing that, on a new object of type object, even if it is an empty object with no properties or functionality.

Synchronisation between threads

We have already seen one type of problem where **multiple threads share resources** and access to these resources is synchronised through the use of monitors. So far, once a thread gets a lock on a monitor, it can make use of it indiscriminately, regardless of any other conditions.

Now we are going to see how, depending on the state of the resources, each thread will be able to perform certain actions or not, allowing the **threads** to **wait** for a **change** of **state** that can be notified by other threads.

For this, in addition to a blocking mechanism on the shared resources, a waiting mechanism will be necessary so that, in the event that the state of the shared resources does not allow a thread to perform an action, the execution of the thread is suspended waiting for that condition to be met.

The mechanism is a **non-active wait** mechanism, i.e. no processor time or system resources should be consumed to check whether it is possible to continue execution, until a **notification** is **received** that the state has changed and could allow the thread to continue execution.

This will also allow us, collaterally, to control the order of execution of the threads depending on the relationship established between them.

To resolve this type of situation, we will be using methods of the **Object class**, accessible to any object.

wait(): **interrupts** the **execution** of the current **thread**. The execution of the thread is **blocked** until another thread executes the **notify()** or **notifyAll()** method on the object. This method therefore provides a non-active wait mechanism.

notify(): **unblocks** one of the threads waiting on an object after having executed the wait() method, so that it can continue its execution. This method provides a notification mechanism to terminate the non-active wait of threads waiting on a blocking object. The **order** in which threads are unblocked on a blocking object is again **indeterministic** and need not match the order in which they were blocked.

notifyAll(): **unblocks all** threads waiting on a lock object after executing the wait() method, so that they can continue their execution.

These **methods** need to be called from **synchronised context**, otherwise they will throw **java.lang.IllegalMonitorStateException**.

When the **wait()** method is called, the thread will be inside a **synchronised block**, therefore it will have the **monitor lock**. At that point the thread releases the lock on that monitor and remains in a queue (belonging to the object) of threads waiting to be notified, different from the queue of threads waiting for the lock.

When a thread is unblocked because another thread has called notify()/notifyAll(), the thread returns to the point where it did the wait(), so it is still inside a synchronised block. In order to continue its execution, it will have to move to the queue of threads waiting for the lock and wait to be selected for further execution.

```
synchronized(blockingObj)
{
  while(conditionToContinue) {
    try {
      // Waits for the condition to change and another thread to warn it
      objBloqueo.wait()
    } catch (InterruptedException e) {}
  }

  // If the thread has reached this point, it means that either at the beginning of the
  // or after having performed one or more waits and been notified
  // of changes by other threads, the condition has been fulfilled.

  //It has also been locked by the monitor so that it can continue.
  // inside the synchronized block
  perform_operation;

  //This part is optional. It can be done by this thread, in this
  //same method, or it can be done by another thread in another method
  if(conditionForOthersToContinue) {
        blockingObj.notify(); // or notifyAll()
  }
}
```

Let's see an example of a simple **multi-threaded application**.

There are two threads **Reader** and **Writer** and both are sharing a common object "**message**".

Reader reads the **message** when it is **not empty** otherwise, it will wait for the **Writer** to **write** the **message**.

On the other hand, **Writer** writes the **message** when it is **empty** else it waits for **Reader** to **read** the **message** and marks the message **empty**.

This is one of the famous problems in multi-threading and is referred to as **Reader-Writer Problem**.

Now let's create two threads **Writer** and **Reader** and one **shared object** for both the thread that is **Message object**.

The **Message class** could be as follows to avoid **deadlock**:

```
public class Message {
    String message;
    boolean empty = true;

    //Method used by reader
    public synchronized String read() {
        while (empty) {
            try {
                /*
                 Reader thread waits until Writer invokes the notify() method or the
notifyAll() method for 'message' object.
```

```
                Reader thread releases ownership of lock and waits until Writer thread
notifies Reader thread waiting on
                this object's lock to wake up either through a call to the notify
method or the notifyAll method.*/
                wait();
            } catch (InterruptedException e) {
                System.out.println(Thread.currentThread().getName() + "Interrupted.");
            }
        }
        empty = true;//Reader reads the message and marks empty as true.
        /*
         Wakes up all threads that are waiting on 'message' object's monitor(lock).
         This thread(Reader) releases the lock for 'message' object.
         */
        notifyAll();
        return message;//Reader reads the message.
    }

    //Method used by writer
    public synchronized void write(String message) {
        while (!empty) {
            try {
                /*
                 Writer thread waits until Reader invokes the notify() method or the
notifyAll() method for 'message' object.
                 Writer thread releases ownership of lock and waits until Reader thread
notifies Writer thread waiting on
                 this object's lock to wake up either through a call to the notify
method or the notifyAll method.*/
                wait();
            } catch (InterruptedException e) {
                System.out.println(Thread.currentThread().getName() + "Interrupted.");
            }
        }
        this.message = message;//Writer writes the message.
        empty = false;//Now make empty as false.
        /*
         Wakes up all threads that are waiting on 'message' object's monitor(lock).
         This thread(Writer) releases the lock for 'message' object.
         */
        notifyAll();
    }
}
```

The **Message** class object message has **two fields** a **string** and a **boolean**. A string holds the message and boolean lets the thread know whether the message variable is empty or not.

The **Writer class** could be the following:

```
import java.util.Random;
public class Writer implements Runnable{
    private Message message;
    public Writer(Message message) {
        this.message = message;
    }

    @Override
    public void run() {
        String messages[] = {
                "The threads can communicate with each other through wait(), notify()
and notifyAll() methods in Java.","These are final methods defined in the Object class
```

```
and can be called only from within a synchronized context.","The wait() method causes
the current thread to wait until another thread invokes the notify() or notifyAll()
methods for that object. ","The notifyAll() method wakes up all threads that are
waiting on that object's monitor.","A thread waits on an object's monitor by calling
one of the wait() method."};
        Random random = new Random();
        for (int i = 0; i < messages.length; i++) {
            message.write(messages[i]);
            try {
                Thread.sleep(random.nextInt(2000));
            } catch (InterruptedException e) {
                System.out.println("Writer Thread Interrupted!!!");
            }
        }
        message.write("Finished!");
    }
}
```

When the **Writer** thread executes the **run** method, it uses its message field to call the **write()** method, which basically is synchronized. Hence, only one thread can execute the write() method on the object message, as the thread executing this method would acquire the lock of the message object. Also, no other method of this message object can be executed by any other threads as there only exists one **intrinsic lock(monitor lock)** per object.

The **Reader class** could be:

```
import java.util.Random;

public class Reader implements Runnable{
    private Message message;

    public Reader(Message message) {
        this.message = message;
    }

    @Override
    public void run() {
        Random random = new Random();
        for (String latestMessage = message.read(); !"Finished!".equals(latestMessage);
latestMessage = message.read()) {
            System.out.println(latestMessage);
            try {
                Thread.sleep(random.nextInt(2000));
            } catch (InterruptedException e) {
                System.out.println("Reader Thread Interrupted!!!");
            }
        }
    }
}
```

When the **Reader** thread executes the **run** method, it also uses the same message object to call the **read()** method, which is also synchronised. Hence, only one thread can execute the read() method on the object message, as the thread executing this method would acquire the lock of the message object. Also, no other method of this message object can be executed by any other threads as there only exists one intrinsic lock per object.

Now let's create instances of **Message**, **Reader**, and **Writer** classes and then start both threads.

```
public class MainReaderWriter {
    public static void main(String[] args) {
        //Shared message object between Reader and Writer threads.
        Message message = new Message();

        Thread writerThread = new Thread(new Writer(message));
        Thread readerThread = new Thread(new Reader(message));

        writerThread.start();
        readerThread.start();
    }
}
```

Producer-consumer problem

The **Producer-Consumer Problem** (sometimes called the **Bounded-Buffer Problem**) is a classic example of a multi-threaded synchronisation problem.

The problem describes two threads, the **Producer**, and the **Consumer**, and they are sharing a common, fixed-size **buffer** that is used as a queue.

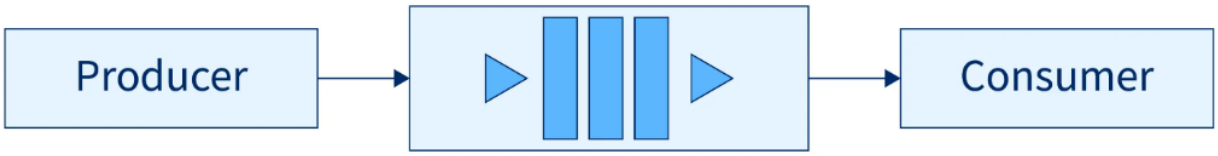
The **Producer** produces an **item**, puts that item into the **buffer**, and keeps repeating this process.

On the other hand, the **Consumer** is **consuming** the item from the shared buffer, one item at a time.

It is necessary to make sure that the **producer** won't try to add data into the **buffer** if it's **full** and that the **consumer** won't try to **remove data** from an **empty buffer**.

The **producer** is to either **go to sleep** or **discard data** if the **buffer** is **full**. The next time the **consumer removes** an **item** from the **buffer**, it **notifies** the **producer**, who starts to **fill** the buffer **again**. In the same way, the **consumer** can **go to sleep** if it finds the **buffer** to be **empty**. The next time the **producer** puts data into the **buffer**, it **wakes up** the **sleeping consumer**.

An **inadequate solution** could result in a **deadlock** where both processes are waiting to be awakened.



The **Producer** thread produces items into the buffer so that the Consumer can consume the items from this buffer.

```
public class Producer implements Runnable{
    Buffer buffer = new Buffer(2);
    public Producer(Buffer buffer){
        this.buffer=buffer;
    }
    @Override
    public void run() {
```

```
        try {
            buffer.produce();
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

The consumer thread consumes items from the buffer produced by the Producer.

```
public class Consumer implements Runnable{
    Buffer buffer = new Buffer(2);
    public Consumer(Buffer buffer){
        this.buffer=buffer;
    }
    @Override
    public void run() {
        try {
            buffer.consume();
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

The buffer class is the shared class and could be:

```
public class Buffer {
    private Queue<Integer> list;
    private int size;

    public Buffer(int size) {
        this.list = new LinkedList<>();
        this.size = size;
    }
    public void produce() throws InterruptedException {
        int value = 0;
        while (true) {
            synchronized (this) {
                while (list.size() >= size) {
                    // wait for the consumer
                    wait();
                }
                list.add(value);
                System.out.println("Produced " + value);
                value++;
                // notify the consumer
                notify();
            }
        }
    }

    public void consume() throws InterruptedException {
        while (true) {
            synchronized (this) {
                while (list.size() == 0) {
                    // wait for the producer
                    wait();
                }
                int value = list.poll();
```

```
                System.out.println("Consume " + value);
                // notify the producer
                notify();
            }
        }
    }
}
```

To test the classes, we can use:

```
public class ProducerConsumerMain {
    public static void main(String[] args) throws InterruptedException {
        Buffer buffer = new Buffer(2);
        Thread producerThread = new Thread(new Producer(buffer));
        Thread consumerThread = new Thread(new Consumer(buffer));
        producerThread.start();
        consumerThread.start();

        producerThread.join();
        consumerThread.join();
    }
}
```