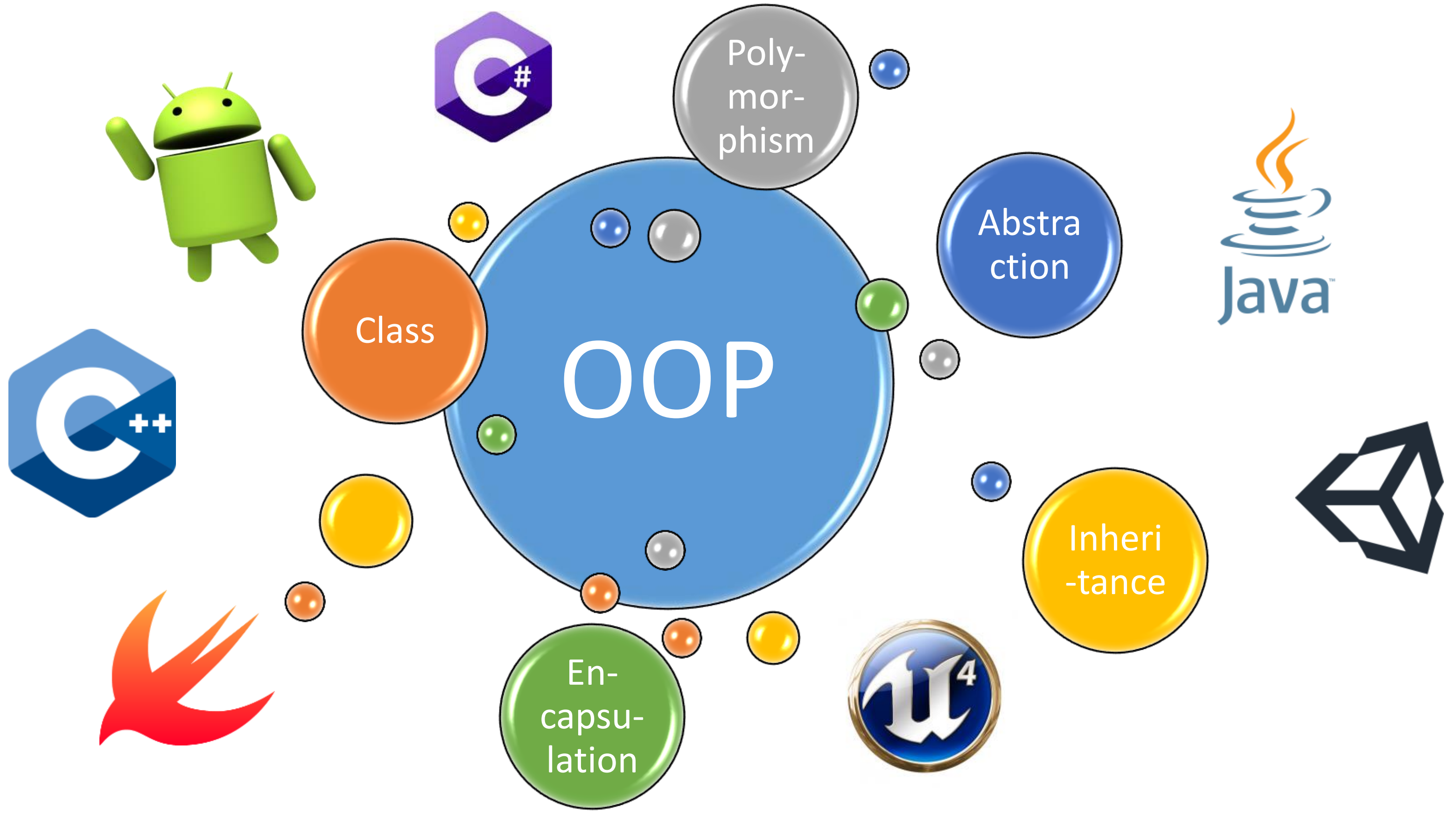
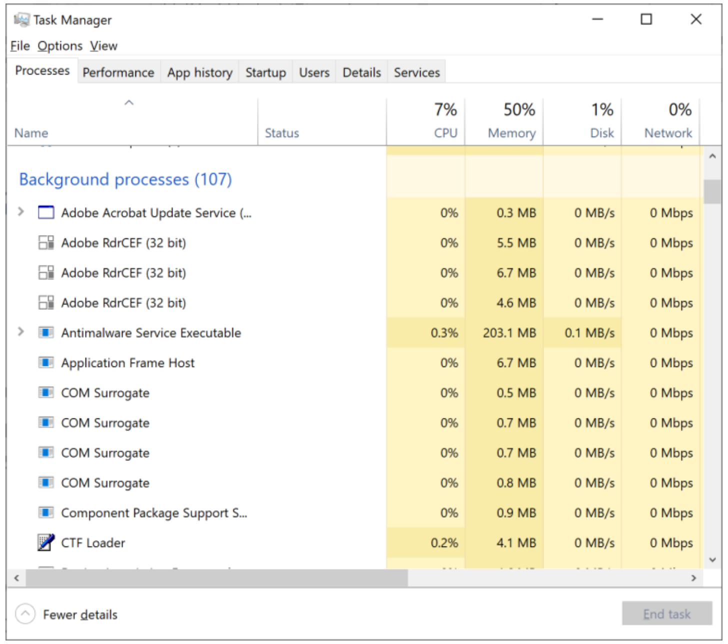


Java Threads

Kuan-Ting Lai
2020/5/16

What is Thread?

- Process vs. Thread
- Process:
  - Any computer program in execution
  - Has independent resources such as memory, file descriptors, security attributes, process state, etc.
- Thread:
  - A component of a process
  - A process can have multiple threads

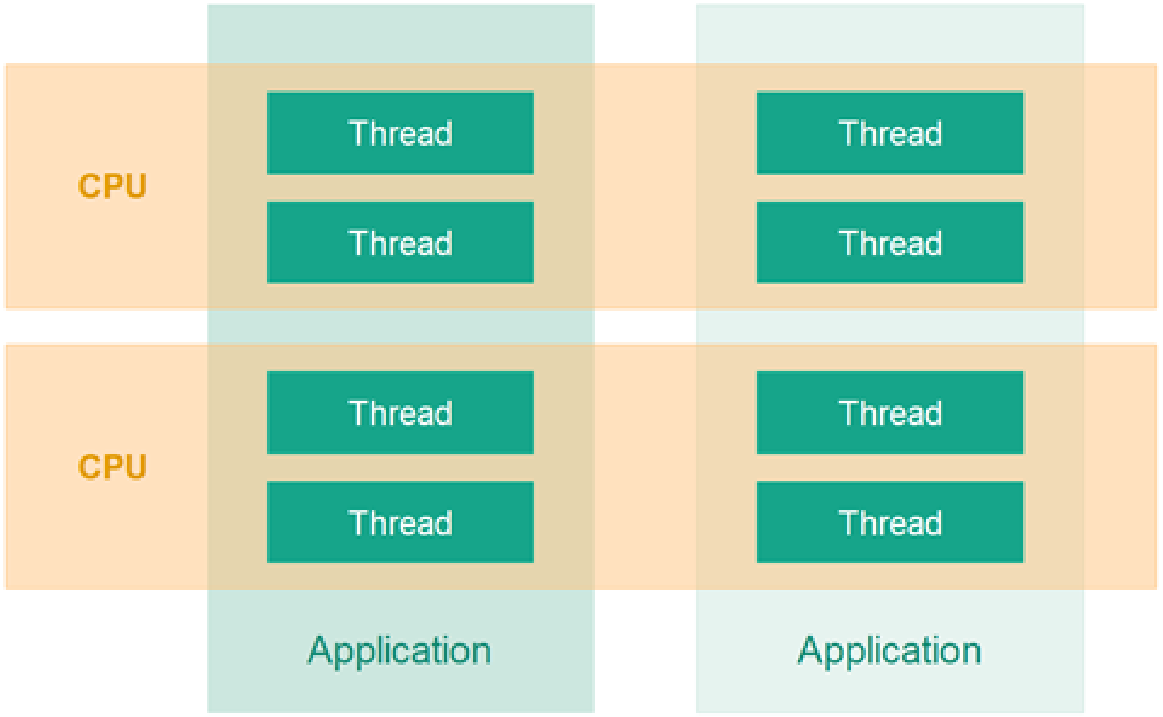


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
End task

Name	Status	7% CPU	50% Memory	1% Disk	0% Network
Background processes (107)					
Adobe Acrobat Update Service (...		0%	0.3 MB	0 MB/s	0 Mbps
Adobe RdrCEF (32 bit)		0%	5.5 MB	0 MB/s	0 Mbps
Adobe RdrCEF (32 bit)		0%	6.7 MB	0 MB/s	0 Mbps
Adobe RdrCEF (32 bit)		0%	4.6 MB	0 MB/s	0 Mbps
Antimalware Service Executable		0.3%	203.1 MB	0.1 MB/s	0 Mbps
Application Frame Host		0%	6.7 MB	0 MB/s	0 Mbps
COM Surrogate		0%	0.5 MB	0 MB/s	0 Mbps
COM Surrogate		0%	0.7 MB	0 MB/s	0 Mbps
COM Surrogate		0%	0.7 MB	0 MB/s	0 Mbps
COM Surrogate		0%	0.8 MB	0 MB/s	0 Mbps
Component Package Support S...		0%	0.9 MB	0 MB/s	0 Mbps
CTF Loader		0.2%	4.1 MB	0 MB/s	0 Mbps

Multithreading

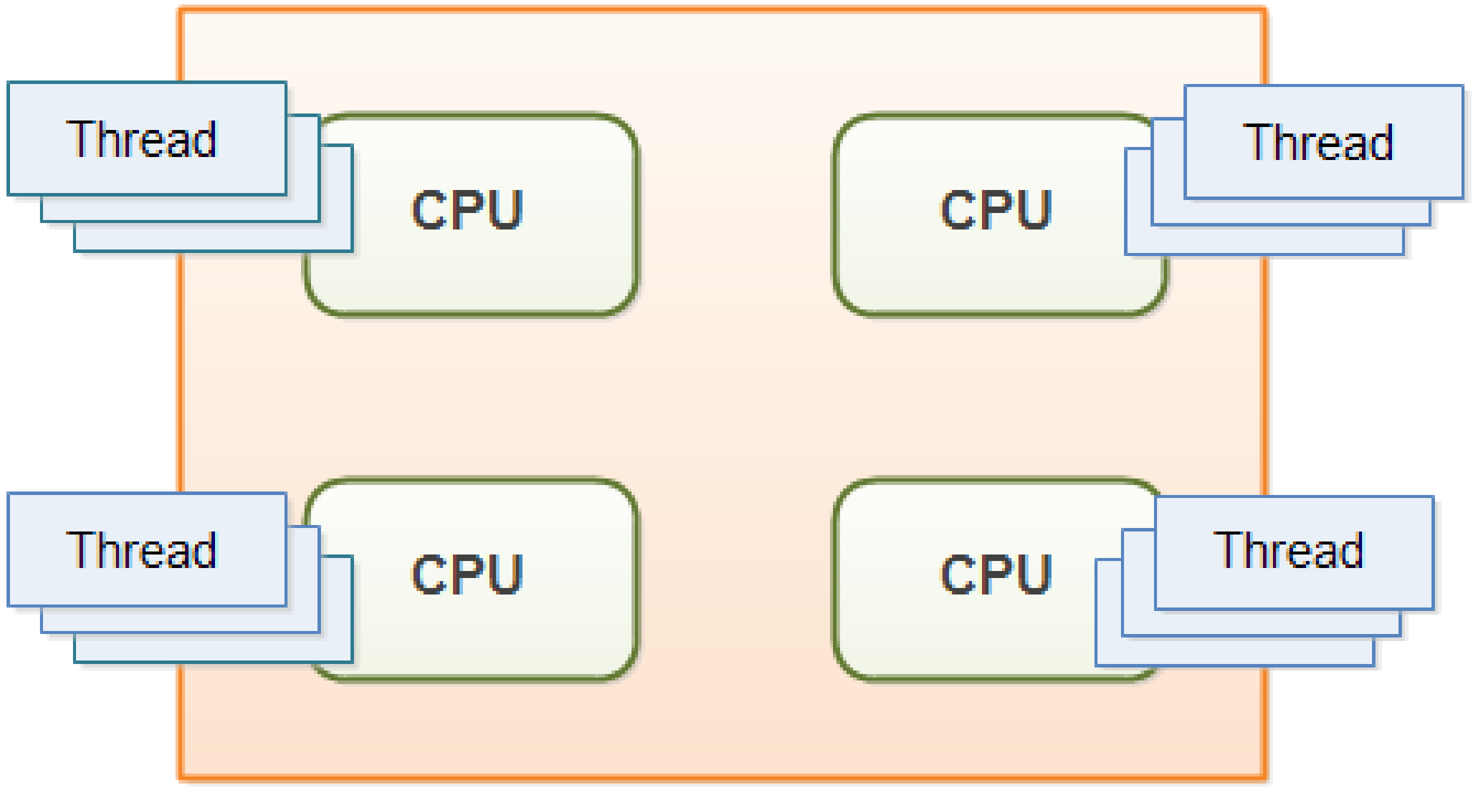
- Share CPU time to multiple threads
- Avoid slow tasks (I/O) occupy CPU time
- Better utilization of a single CPU
- Better user responsiveness



http://tutorials.jenkov.com/java-concurrency/index.html

Multithreading is Hard

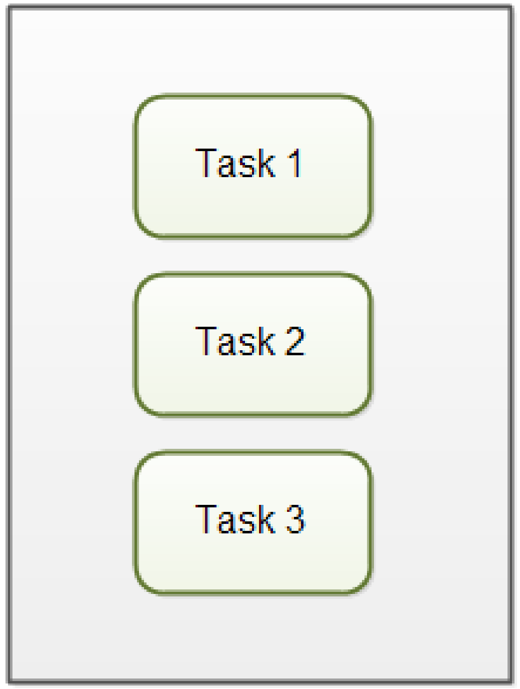
- Threads may access resources simultaneously



Concurrency vs. Parallelism

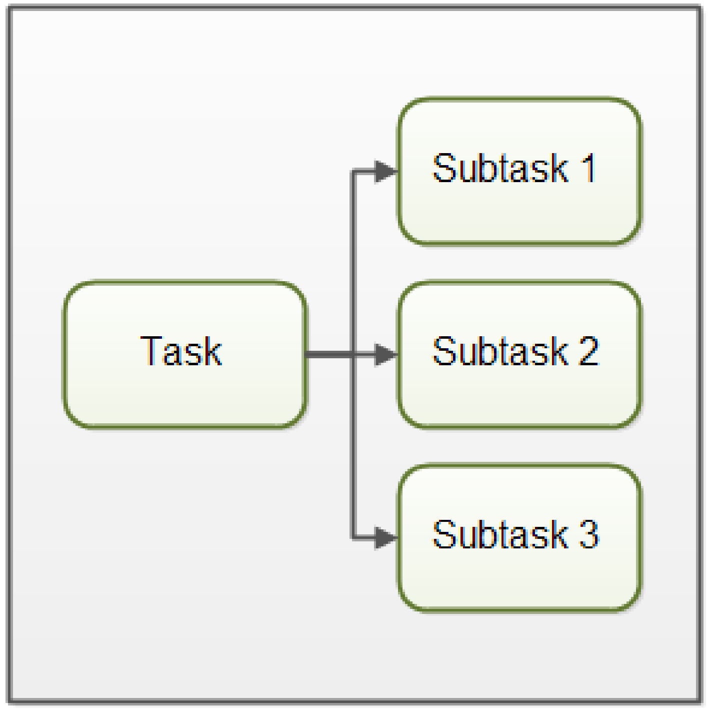
Concurrency

- Running (switching) multiple tasks at the same time

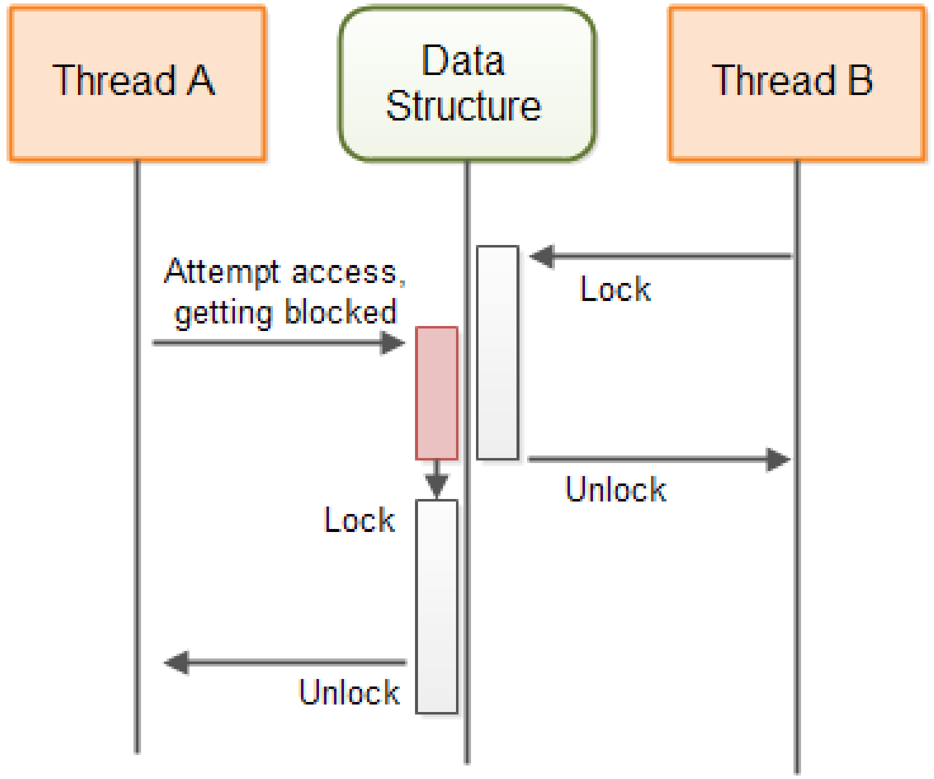


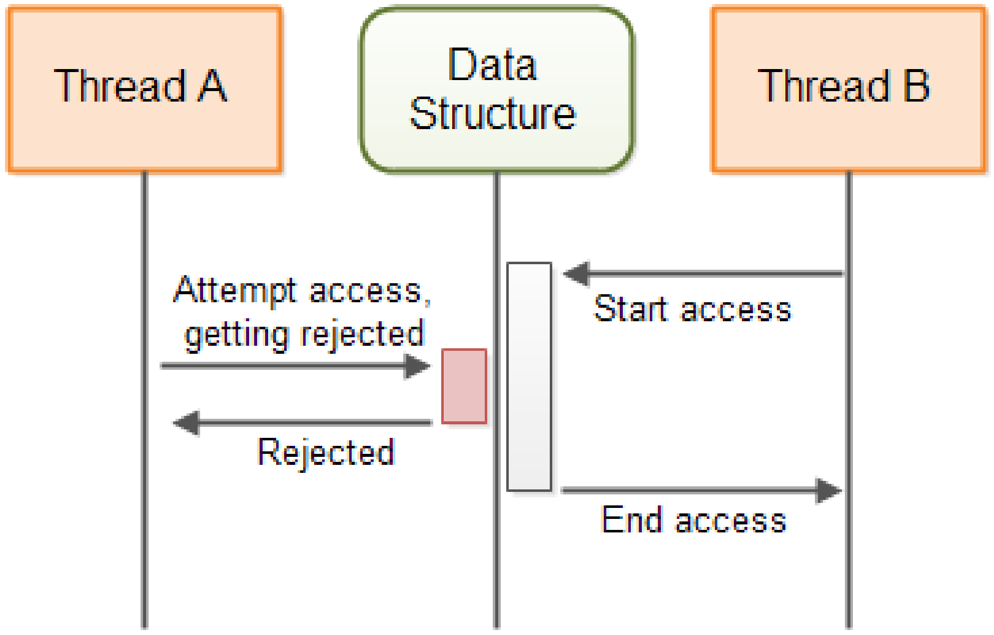
Parallelism

- Divide tasks into individual subtasks



Blocking vs. Non-Blocking

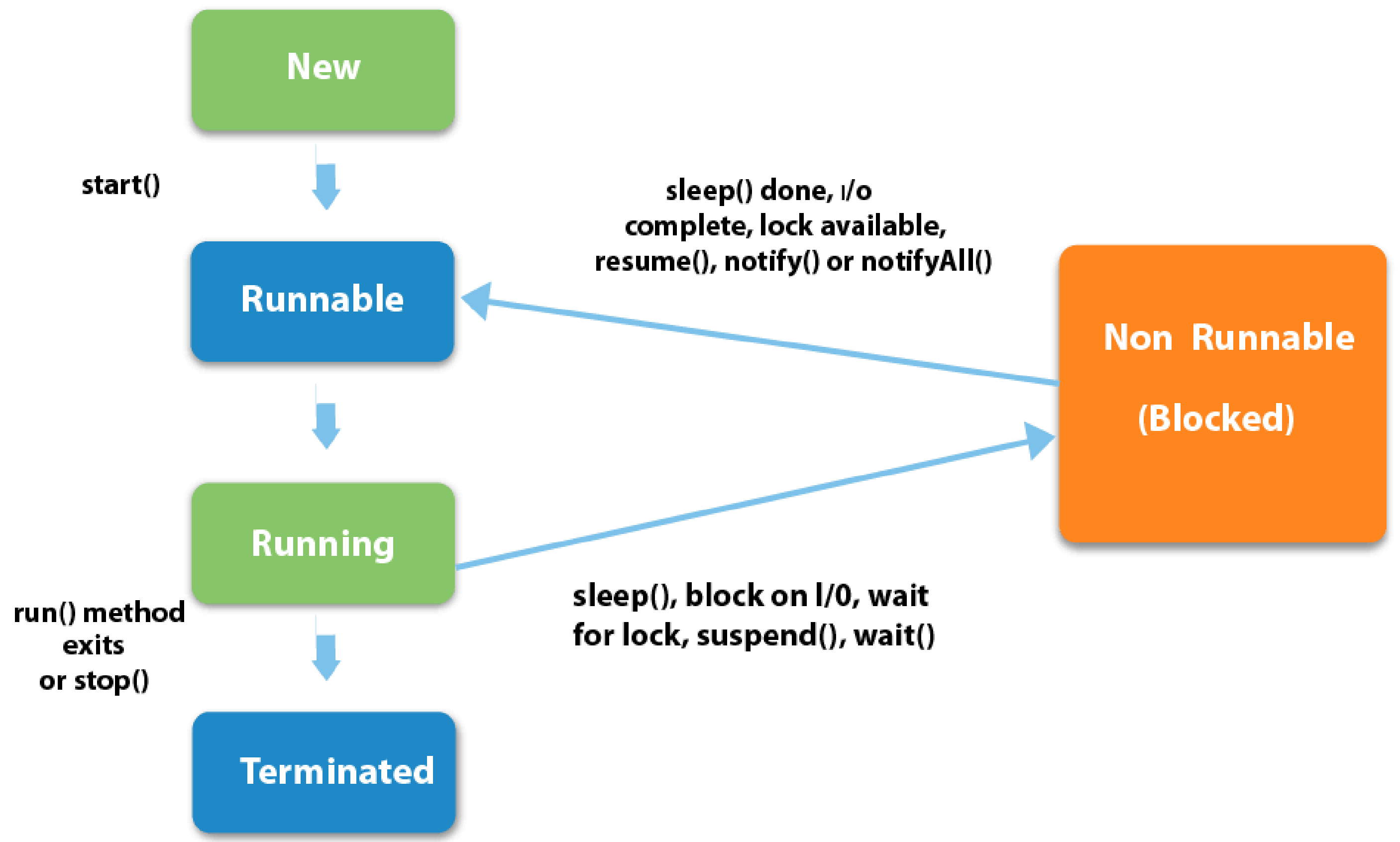




http://tutorials.jenkov.com/java-concurrency/non-blocking-algorithms.html

Java Thread Lifecycle

- NEW
  - A thread that has not yet started
- RUNNABLE
  - A thread executing in the JVM
- BLOCKED
  - Waiting for a monitor lock
- WAITING
  - Waiting indefinitely for another thread to perform an action
- TIMED_WAITING
  - Waiting for up to a specified waiting time
- TERMINATED


New
start()
Runnable
sleep() done, I/o complete, lock available, resume(), notify() or notifyAll()
Non Runnable
(Blocked)
Running
sleep(), block on I/0, wait for lock, suspend(), wait()
run() method exits or stop()
Terminated

Creating a Thread (1)

- Inherit Thread class

```
public class MyThread extends Thread {
    public void run() {
        System.out.println("MyThread running");
    }
}
MyThread myThread = new MyThread();
myTread.start();
```

Creating a Thread (2)

- Implement Runnable interface

```java
public class MyRunnable implements Runnable {
    public void run() {
        System.out.println("MyRunnable running");
    }
}
Runnable runnable = new MyRunnable(); // or an anonymous class, or lambda...
Thread thread = new Thread(runnable);
thread.start();
```

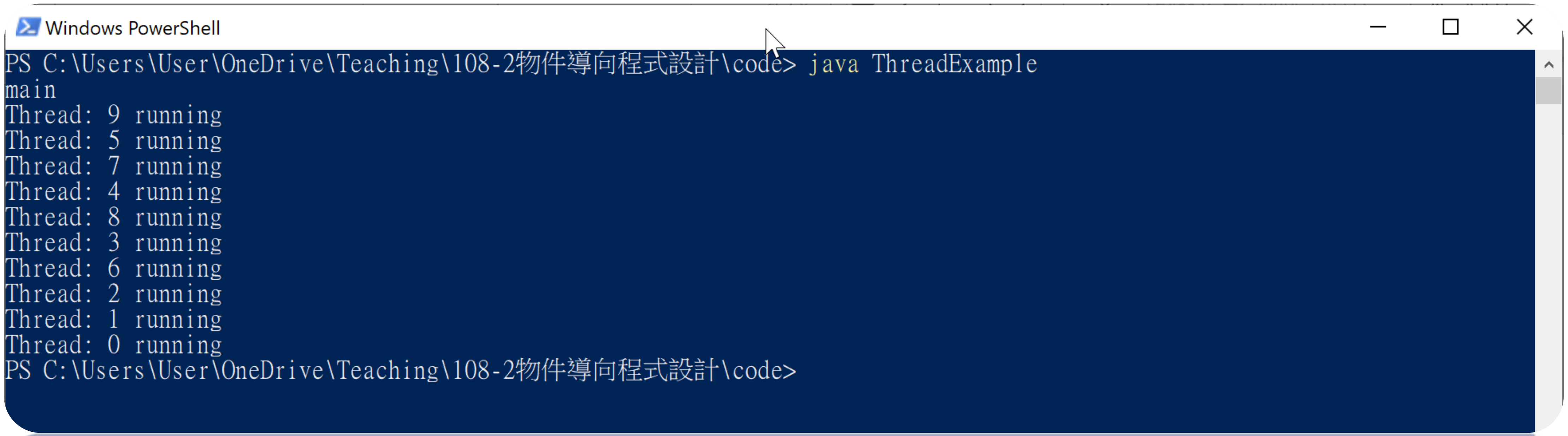
Thread Example

- Create 10 threads with serial ID (0 ~ 9)

```
public class ThreadExample {

    public static void main(String[] args) {
        System.out.println(Thread.currentThread().getName());
        for (int i = 0; i < 10; i++) {
            new Thread("" + i){
              public void run() {
               System.out.println("Thread: " + getName() + " running");
              }
            }.start();
        }
    }
}
```

Running ThreadExample

- Threads are not executed sequentially!



```
PS C:\Users\User\OneDrive\Teaching\108-2物件導向程式設計\code> java ThreadExample
main
Thread: 9 running
Thread: 5 running
Thread: 7 running
Thread: 4 running
Thread: 8 running
Thread: 3 running
Thread: 6 running
Thread: 2 running
Thread: 1 running
Thread: 0 running
PS C:\Users\User\OneDrive\Teaching\108-2物件導向程式設計\code>
```

Thread.sleep

- Pause a thread for 10 second

```
try {
      Thread.sleep(10L * 1000L);
}
catch (InterruptedException e) {
      e.printStackTrace();
}
```

Stop a Thread

```
public class MyRunnable implements Runnable {

    private boolean isStop = false;

    public synchronized void doStop() {
        this.isStop = true;
    }

    @Override
    public void run() {
        while (!this.isStop) {
            System.out.println("Running");
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Race Condition and Critical Sections

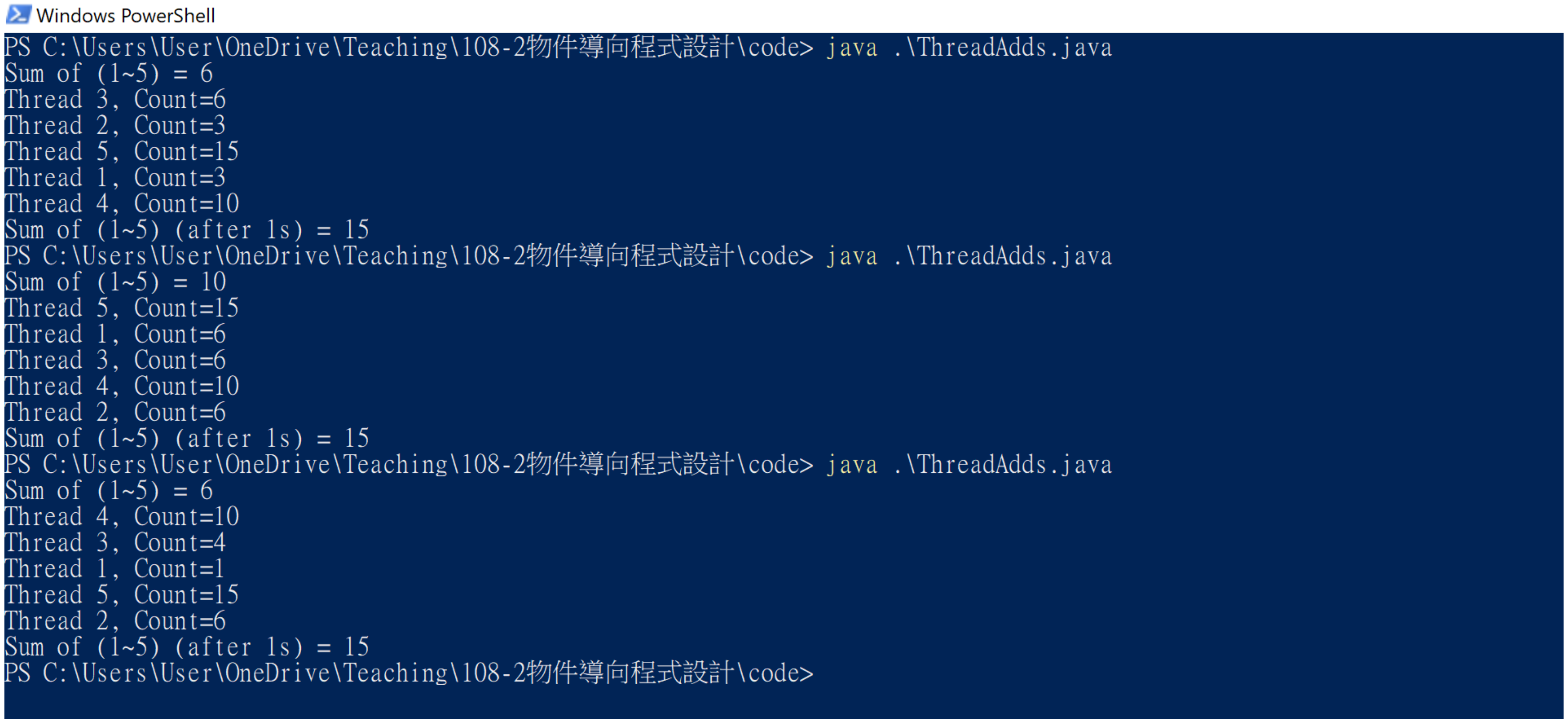
- Race condition is caused when two threads are writing the same memory
- Use critical section synchronized to protect the area

```
public class TwoSums {
    private int sum1 = 0;
    private int sum2 = 0;

    public void add(int val1, int val2) {
        synchronized(this) {
            this.sum1 += val1;
            this.sum2 += val2;
        }
    }
}
```

Race Condition Example

```java
public class Counter {
    public int x = 0;
    public void add(int value) {
        x += value;
        System.out.println("Thread " + value + ", Count=" + x);
    }
}
public class CounterThread implements Runnable {
    int target = 0;
    Counter count;
    CounterThread(Counter vptr, int var) {
        count = vptr; target = var;
    }
    public void run() {
        count.add(target);
    }
}
public class ThreadAdds {
    public static void main(String[] args) {
        Counter count = new Counter();
        for (int i = 1; i <= 5; i++) {
                new Thread(new CounterThread(count, i)).start();
        }
        System.out.println("Sum of (1~5) = " + count.x);
        try {
            Thread.sleep(1000);
            System.out.println("Sum of (1~5) (after 1s) = " + count.x);
        }
        catch (InterruptedException e) {
                e.printStackTrace();
        }}}
```

Race Condition Results

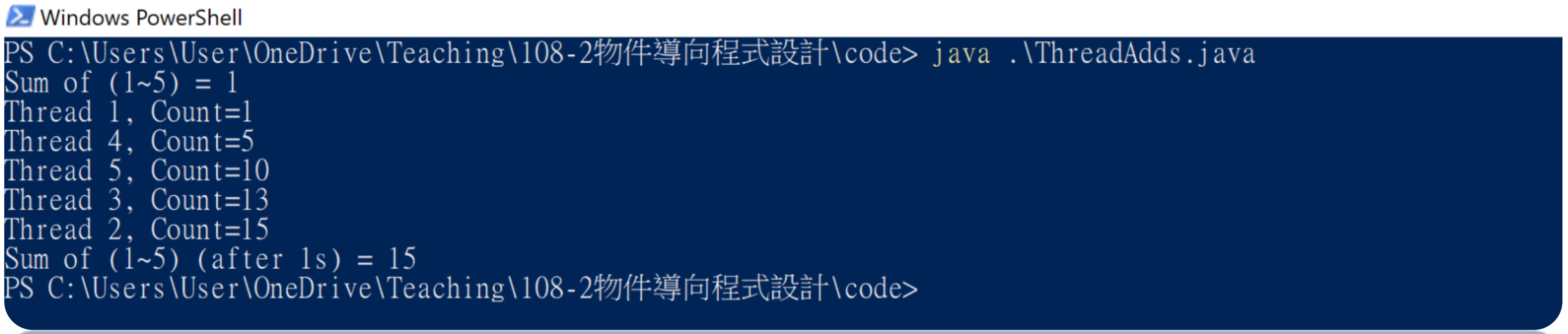


Windows PowerShell

```
PS C:\Users\User\OneDrive\Teaching\108-2物件導向程式設計\code> java .\ThreadAdds.java
Sum of (1~5) = 6
Thread 3, Count=6
Thread 2, Count=3
Thread 5, Count=15
Thread 1, Count=3
Thread 4, Count=10
Sum of (1~5) (after 1s) = 15
PS C:\Users\User\OneDrive\Teaching\108-2物件導向程式設計\code> java .\ThreadAdds.java
Sum of (1~5) = 10
Thread 5, Count=15
Thread 1, Count=6
Thread 3, Count=6
Thread 4, Count=10
Thread 2, Count=6
Sum of (1~5) (after 1s) = 15
PS C:\Users\User\OneDrive\Teaching\108-2物件導向程式設計\code> java .\ThreadAdds.java
Sum of (1~5) = 6
Thread 4, Count=10
Thread 3, Count=4
Thread 1, Count=1
Thread 5, Count=15
Thread 2, Count=6
Sum of (1~5) (after 1s) = 15
PS C:\Users\User\OneDrive\Teaching\108-2物件導向程式設計\code>
```

Using Critical Section (synchronized)

```java
public class Counter {
    public int x = 0;
    public synchronized void add(int value) {
        x += value;
        System.out.println("Thread " + value + ", Count=" + x);
    }
}
```



Windows PowerShell

```
PS C:\Users\User\OneDrive\Teaching\108-2物件導向程式設計\code> java .\ThreadAdds.java
Sum of (1~5) = 1
Thread 1, Count=1
Thread 4, Count=5
Thread 5, Count=10
Thread 3, Count=13
Thread 2, Count=15
Sum of (1~5) (after 1s) = 15
PS C:\Users\User\OneDrive\Teaching\108-2物件導向程式設計\code>
```

Synchronized Block

- Lock only part of the code

```
public class Counter {
    public int x = 0;

    public void add(int value) {
        synchronized(this) {
            x += value;
            System.out.println("Thread " + value + ", Count=" + x);
        }
    }
}
```

Thread Safety and Immutability

- Member variables are not writable

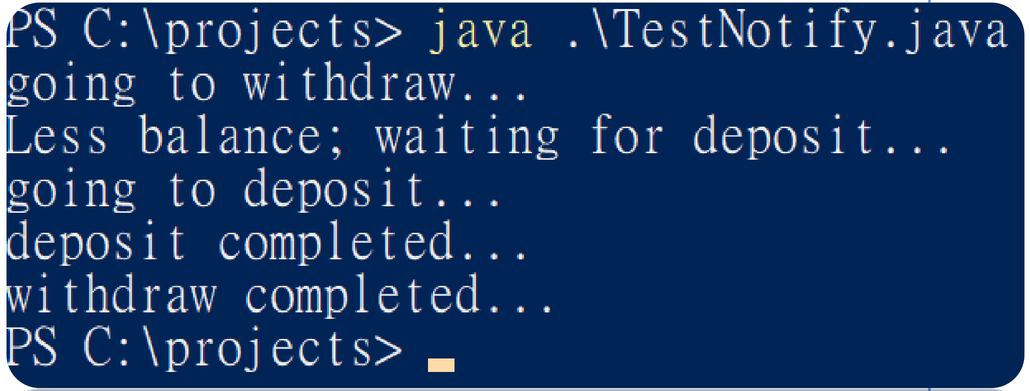
```
public class ImmutableValue {

    private int value = 0;

    public ImmutableValue(int value) {
        this.value = value;
    }

    public int getValue() {
        return this.value;
    }
}
```

Thread Signaling

- Enable threads to send signals to each other
- Traditional methods
  - shared variables, busy wait
- Java approach
  - Object methods: wait(), notify() and notifyAll()

http://tutorials.jenkov.com/java-concurrency/thread-signaling.html

```java
class Customer {
    int amount = 10000;
    synchronized void withdraw(int amount) {
        System.out.println("going to withdraw...");
        if (this.amount < amount) {
            System.out.println("Less balance; waiting for deposit...");
            try { wait(); }
            catch (Exception e) {}
        }
        this.amount -= amount;
        System.out.println("withdraw completed...");
    }
    synchronized void deposit(int amount) {
        System.out.println("going to deposit...");
        this.amount += amount;
        System.out.println("deposit completed... ");
        notify();
    }
}
class TestNotify {
    public static void main(String args[]) {
        final Customer c = new Customer();
        new Thread(){ public void run() { c.withdraw(15000); } }.start();
        new Thread(){ public void run() { c.deposit(10000); }}.start();
    }
}
```



```
PS C:\projects> java .\TestNotify.java
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed...
PS C:\projects> _
```

Producer and Consumer

- Producer thread produces a new resource in every 1 second and put it in “taskQueue”
- Consumer thread takes 1 second to process consumed resource from “taskQueue”
- Max capacity of taskQueue is 5 resources
- Both threads run infinitely

https://howtodoinjava.com/java/multi-threading/wait-notify-and-notifyall-methods/

Producer

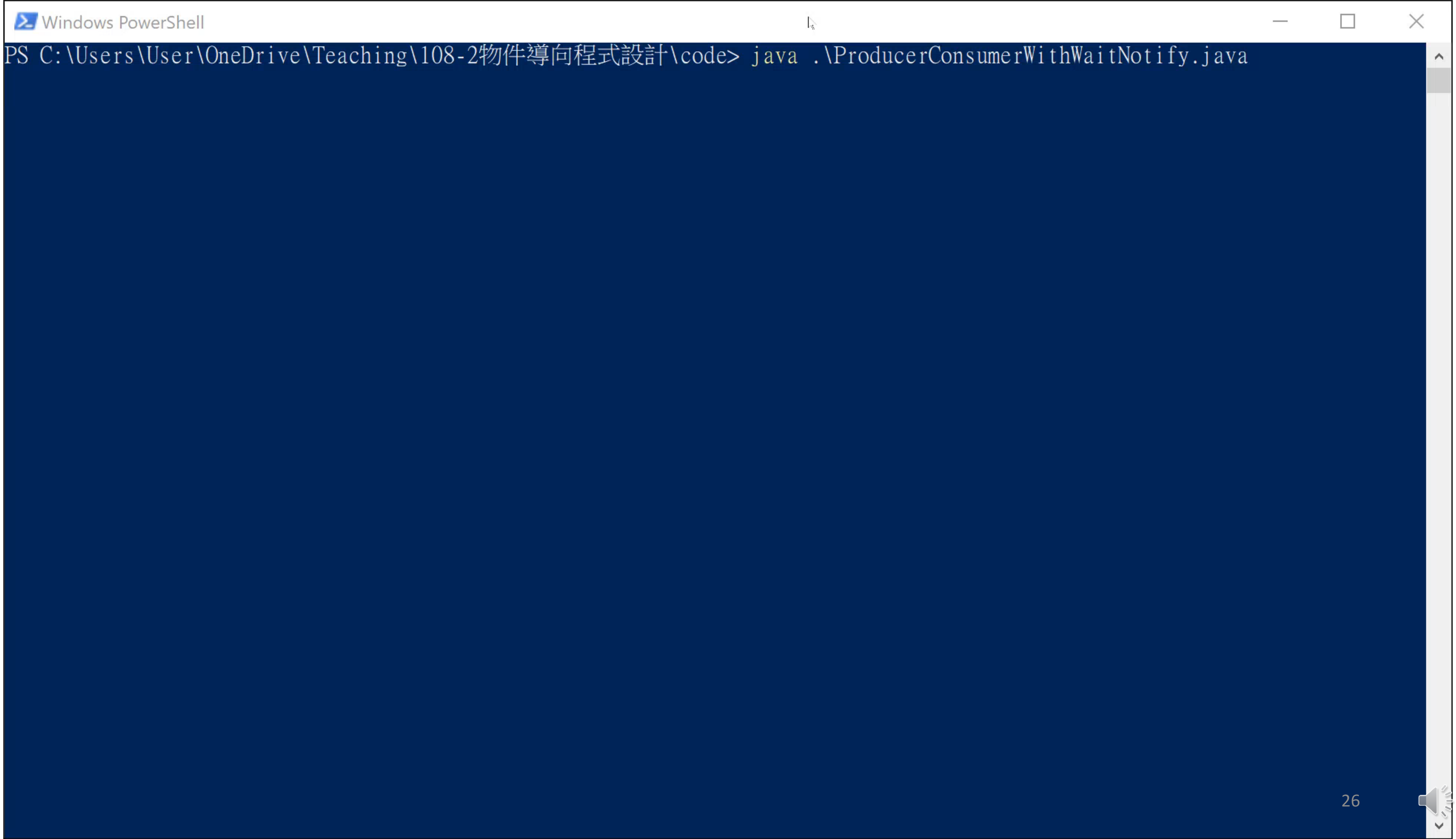
```
class Producer implements Runnable {
    private final List<Integer> taskQueue;
    private final int MAX_CAPACITY;
    public Producer(List<Integer> sharedQueue, int size) {
        this.taskQueue = sharedQueue;
        this.MAX_CAPACITY = size;
    }
    public void run() {
        int counter = 0;
        while (true) {
            try { produce(counter++);
            } catch (InterruptedException ex) {
                ex.printStackTrace();}
        }
    }
    private void produce(int i) throws InterruptedException {
        synchronized(taskQueue) {
            while (taskQueue.size() == MAX_CAPACITY) {
                System.out.println("Queue is full " + Thread.currentThread().getName() +
                " is waiting , size: " + taskQueue.size());
                taskQueue.wait();
            }
            Thread.sleep(1000);
            taskQueue.add(i);
            System.out.println("Produced: " + i);
            taskQueue.notifyAll();
        }}}
```

Consumer

```java
class Consumer implements Runnable {
    private final List<Integer> taskQueue;
    public Consumer(List<Integer> sharedQueue) {
        this.taskQueue = sharedQueue;
    }
    public void run() {
        while (true) {
            try { consume();
            } catch (InterruptedException ex) {
                ex.printStackTrace();}
        }
    }
    private void consume() throws InterruptedException {
        synchronized(taskQueue) {
            while (taskQueue.isEmpty()) {
                System.out.println("Queue is empty " + Thread.currentThread().getName()
                + " is waiting , size: " + taskQueue.size());
                taskQueue.wait();
            }
            Thread.sleep(1000);
            int i = (Integer)taskQueue.remove(0);
            System.out.println("Consumed: " + i);
            taskQueue.notifyAll();
        }}}
```

Main Function of ProducerConsumer

```
import java.util.*

public class ProducerConsumerWithWaitNotify
{
    public static void main(String[] args)
    {
        List<Integer> taskQueue = new ArrayList<Integer>();
        int MAX_CAPACITY = 5;
        Thread tProducer = new Thread(new Producer(taskQueue, MAX_CAPACITY), "Producer");
        Thread tConsumer = new Thread(new Consumer(taskQueue), "Consumer");
        tProducer.start();
        tConsumer.start();
    }
}
```



Windows PowerShell

```
PS C:\Users\User\OneDrive\Teaching\108-2物件導向程式設計\code> java .\ProducerConsumerWithWaitNotify.java
```

The join() method

```
class TestJoinMethod1 extends Thread {
    public void run() {
        for (int i = 1; i <= 5; i++) {
            try {
                Thread.sleep(100);
            }
            catch (Exception e) { System.out.println(e); }
            System.out.println(i);
        }
    }
    public static void main(String args[]) {
        TestJoinMethod1 t1 = new TestJoinMethod1();
        TestJoinMethod1 t2 = new TestJoinMethod1();
        TestJoinMethod1 t3 = new TestJoinMethod1();
        t1.start();
        try {
                t1.join();
        }
        catch (Exception e) { System.out.println(e); }
        t2.start();
        t3.start();
    }
}
```

https://www.javatpoint.com/join()-method

Thread Priority

- MIN_PRIORITY, NORM_PRIORITY, MAX_PRIORITY

```
class TestMultiPriority1 extends Thread {
    public void run() {
        System.out.println("Thread name: " + Thread.currentThread().getName());
        System.out.println("Thread priority: " + Thread.currentThread().getPriority());
    }
    public static void main(String args[]) {
        TestMultiPriority1 m1 = new TestMultiPriority1();
        TestMultiPriority1 m2 = new TestMultiPriority1();
        m1.setPriority(Thread.MIN_PRIORITY);
        m2.setPriority(Thread.MAX_PRIORITY);
        m1.start();
        m2.start();
    }
}
```

https://www.javatpoint.com/priority-of-a-thread

Thread Synchronization

- Mutual Exclusive
  - Synchronized method.
  - Synchronized block.
  - static synchronization.
- Cooperation (Inter-thread communication in java)
  - wait(), notify(), notifyAll()

Deadlock

- Two threads are waiting for locks of each other
  - Thread 1 locks A, waits for B
  - Thread 2 locks B, waits for A

https://nedbatchelder.com/blog/200801/deadlock_in_real_life.html

Deadlock Example

```
public class TestDeadlockExample1 {
    public static void main(String[] args) {
        final String resource1 = "ratan jaiswal";
        final String resource2 = "vimal jaiswal";
        // t1 tries to lock resource1 then resource2
        Thread t1 = new Thread(){
            public void run() {
                synchronized(resource1) {
                    System.out.println("Thread 1: locked resource 1");
                    try { Thread.sleep(100); }
                    catch (Exception e) {}
                    synchronized(resource2) {
                    System.out.println("Thread 1: locked resource 2");
                }}}};
        // t2 tries to lock resource2 then resource1
        Thread t2 = new Thread(){
            public void run() {
                synchronized(resource2) {
                    System.out.println("Thread 2: locked resource 2");
                    try { Thread.sleep(100); }
                    catch (Exception e) {}
                    synchronized(resource1) {
                    System.out.println("Thread 2: locked resource 1");
                }}}};
        t1.start();
        t2.start();
    }
}
```

https://www.javatpoint.com/deadlock-in-java

Preventing Deadlock

- Lock timeout
  - `wait(1000/*ms*/)`
- Lock ordering

```
Thread 1:
  lock A
  lock B

Thread 2:
   wait for A
   lock C (when A locked)

Thread 3:
   wait for A
   wait for B
   wait for C
```

Starvation and Fairness

- Threads with high priority occupy all CPU time from threads with lower priority
- Threads are blocked indefinitely waiting to enter a synchronized block
- Threads are waiting on an object indefinitely (called wait())

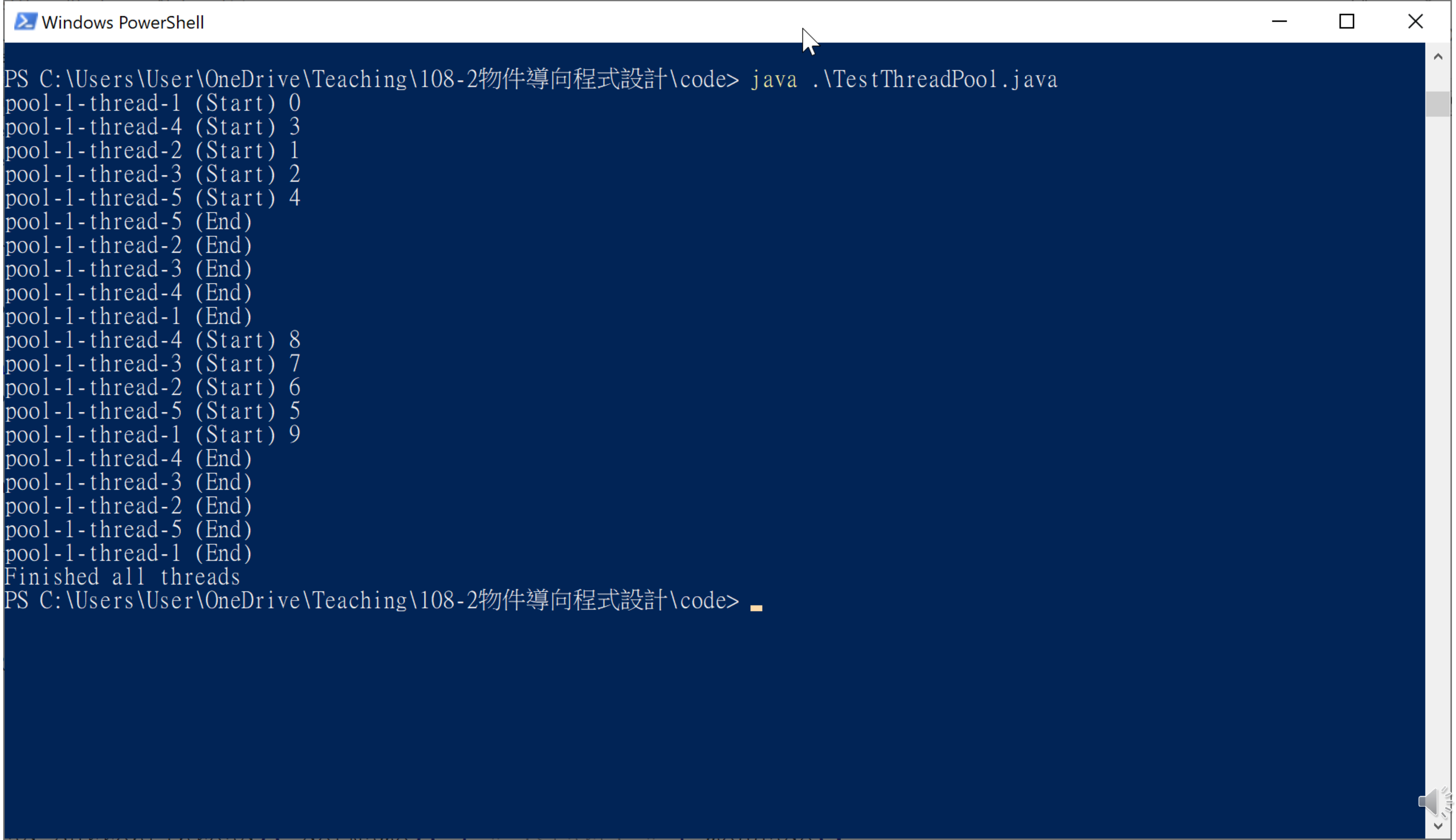
Java Thread Pool

- A group of worker threads that are waiting for the job and reuse many times.

```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
class WorkerThread implements Runnable {
    private String message;
    public WorkerThread(String s) {
        this.message = s;
    }
    public void run() {
        System.out.println(Thread.currentThread().getName() + " (Start) " + message);
        processmessage(); // sleeps the thread for 2 seconds
        System.out.println(Thread.currentThread().getName() + " (End)");
    }
    private void processmessage() {
        try { Thread.sleep(2000); }
        catch (InterruptedException e) { e.printStackTrace(); }
    }
}
```

Test Thread Pool

```java
public class TestThreadPool {
    public static void main(String[] args) {
        //creating a pool of 5 threads
        ExecutorService executor = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 10; i++) {
            Runnable worker = new WorkerThread("" + i);
            executor.execute(worker);//calling execute method of ExecutorService
        }
        executor.shutdown();
        while (!executor.isTerminated()) {}

        System.out.println("Finished all threads");
    }
}
```



```
PS C:\Users\User\OneDrive\Teaching\108-2物件導向程式設計\code> java .\TestThreadPool.java
pool-1-thread-1 (Start) 0
pool-1-thread-4 (Start) 3
pool-1-thread-2 (Start) 1
pool-1-thread-3 (Start) 2
pool-1-thread-5 (Start) 4
pool-1-thread-5 (End)
pool-1-thread-2 (End)
pool-1-thread-3 (End)
pool-1-thread-4 (End)
pool-1-thread-1 (End)
pool-1-thread-4 (Start) 8
pool-1-thread-3 (Start) 7
pool-1-thread-2 (Start) 6
pool-1-thread-5 (Start) 5
pool-1-thread-1 (Start) 9
pool-1-thread-4 (End)
pool-1-thread-3 (End)
pool-1-thread-2 (End)
pool-1-thread-5 (End)
pool-1-thread-1 (End)
Finished all threads
PS C:\Users\User\OneDrive\Teaching\108-2物件導向程式設計\code>
```

Recap

- Thread & Runnable
- Race Condition & Critical Section
- synchronized
- wait(), notify(), notifyAll()
- Deadlock
- Thread Pool

Reference

- http://tutorials.jenkov.com/java-concurrency/creating-and-starting-threads.html
- https://www.javatpoint.com/multithreading-in-java
- https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html
- https://howtodoinjava.com/java/multi-threading/