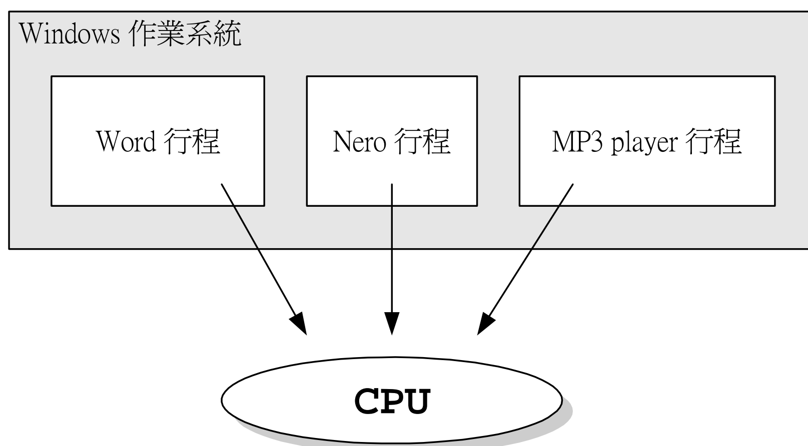
Chapter 16 – Multithreading

- **Life Cycle of a Thread**
- **Thread Priorities and Scheduling**
- **Creating and Executing Threads**
- Runnable **Interface**
- **Thread Synchronization**
- **Producer/Consumer with/without Synchronization**
- **Producer/Consumer Synchronization: Circular Buffer**
- **Daemon Threads**

1

多個行程使用單一CPU



2

行程

- 不同的行程所佔有的記憶體資源不同，各自獨立互不干擾。

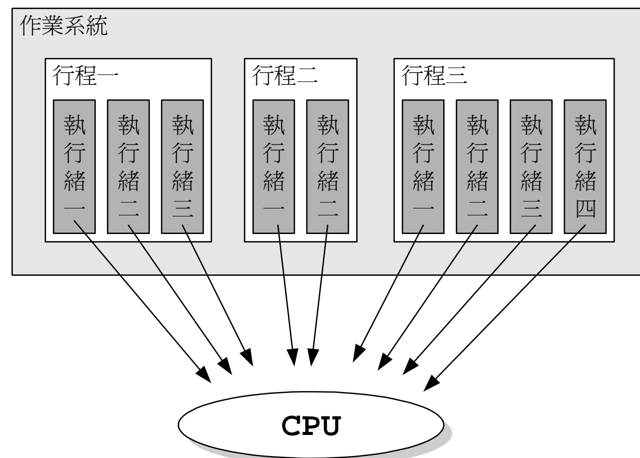
  大部份的個人電腦只有一個CPU，所有行程都必須透過系統取得CPU的使用權。

  每個行程輪流使用CPU的情形就像是每個行程都同時執行一樣。

3

執行緒

多行程多執行緒使用單一CPU



4

行程

- 執行緒是行程中的子程式，這些子程式共享行程內的資源，同時也可以分配到CPU的使用權。
- main()其實就是Java應用程式的預設執行緒（或稱主執行緒，Main Thread）。
- 多執行緒程式必須靠主執行緒去啟動其它執行緒的進行。
- Java的執行緒都必須是java.lang.Thread類別的物件。

5

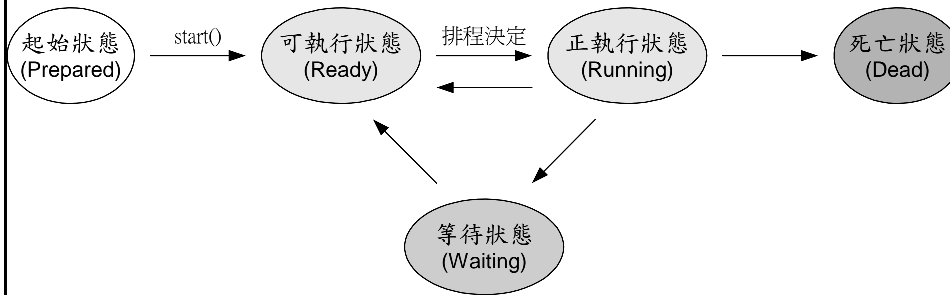
Introduction

- Concurrency normally available in OS primitives
- Java provides built-in multithreading
  - Multithreading improves the performance of some programs

6

Thread States: Life Cycle of a Thread

- 執行緒的狀態包括起始狀態（Prepared）、可執行狀態（Ready）、正執行狀態（Running）、等待狀態（Waiting）及死亡狀態（Dead）。



7

Thread States: Life Cycle of a Thread

- 起始狀態(Prepared)：當我們使用new關鍵字建立一個Thread物件後，未呼叫其start()方法。
- 可執行狀態(Ready)：正等著 thread 排程者安排執行，呼叫其start()方法。
- 正執行狀態(Running)：執行緒正在使用CPU的狀態，正在執行Run ()方法。
- 等待狀態(Waiting)：等待某事件的發生才繼續的狀態，例如呼叫sleep()方法。
- 死亡狀態(Dead)：執行緒執行完成後即進入死亡狀態。進入死亡狀態的執行緒不能被重新啟動。

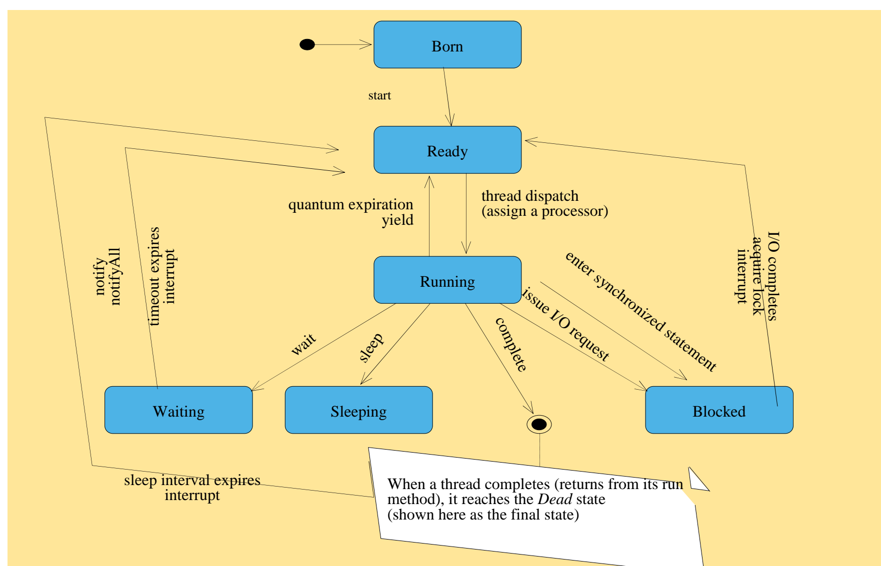
8

Thread States: Life Cycle of a Thread

- Thread states
  - Born state
    - Thread was just created
  - Ready state
    - Thread's **start()** method invoked
    - Thread can now execute
  - Running state
    - Thread is assigned a processor and running execute **run()** method
  - Dead state
    - Thread has completed or exited ➔ **stop() or destroy()** method
    - Eventually disposed of by system

9

Thread life-cycle statechart diagram



10

等待狀態

- 執行sleep()方法：暫停執行緒一段時間。
- 呼叫join()方法：呼叫目標執行緒的join()方法，將等到目標執行緒結束之後才會繼續。
- Input/Output blocked：執行緒進行輸入輸出時，會因為輸入輸出的速度較慢而暫時進入Waiting。
- 呼叫同步方法時，未取得物件的lock。
- 執行wait()方法：放棄物件的使用權並進入等待狀態，可使用notify()方法喚醒執行緒。

11

Thread的建構子及啟動、暫停方法

Thread之建構子及方法	說明
Thread(String name)	建立執行緒，同時設定執行緒的名稱為name。
final String **getName**()	取得執行緒的名稱。
static void **sleep**(long ms) throws InterruptedException	讓執行緒暫停ms毫秒的時間（若暫停的期間執行緒被中斷，則會丟出InterruptedException例外）。
void **run**()	執行緒啟動時所執行的方法。
void **start**()	讓執行緒啟動的方法。

12

Thread的建構子及啟動、暫停方法

- 欲啟動一個執行緒時，必須呼叫它的start()方法，而不是直接呼叫它的run()方法。
- 若直接呼叫Thread類別物件的run()方法，並不會啟動執行緒，只是一般方法的呼叫。
- 執行緒結束之後，執行緒物件還存在，但是不可以重新啟動。

13

Runnable介面

- 執行緒欲繼承Thread以外的類別時，可以利用Runnable介面建立。
- Runnable介面只宣告一個run()方法，所以實作介面時只要實作run()方法即可。
- 實作Runnable介面的類別，還是必須依賴Thread類別的建構子才能建立一個執行緒物件。

使用Runnable介面的優點是不需要繼承Thread，而可以使用執行緒的功能

14

Runnable介面

- 使用Runnable型別物件建立Thread物件的建構子

使用Runnable物件的Thread建構子	說明
Thread(Runnable target)	以實作Runnable介面的類別物件target建立執行緒物件。
Thread(Runnable target, String name)	以實作Runnable介面的類別物件target建立執行緒物件，執行緒的名稱設定為name。

Runnable介面

- 使用Runnable型別物件建立Thread物件，兩個物件是獨立的，不過可以看成是「Thread物件在啟動之後，呼叫Runnable型別物件的run()方法」。

```
final RunnableTest rt = new RunnableTest();
Thread mt = new Thread(){
        public void run() { rt.run(); }
};
```

Thread Priorities and Thread Scheduling

- Java thread priority
  - Priority in range 1-10
- Timeslicing
  - Each thread assigned time on the processor (called a quantum)
  - Keeps highest priority threads running

17

Thread Priorities and Thread Scheduling

- Java的多執行緒排程是使用簡單的「固定優先權排程」(fixed priority scheduling)，這種排程會依據可執行狀態之執行緒的優先權來決定順序。
- 子執行緒擁有和父執行緒相同的優先權。
- 執行緒建立之後，就可以使用getPriority()取得優先權值，以setPriority()方法來設定優先權值。
- Thread類別中定義了三個類別常數，NORM_PRIORITY是預設的優先權值，MIN_PRIORITY是最小的優先權值，MAX_PRIORITY為最大的優先權值。

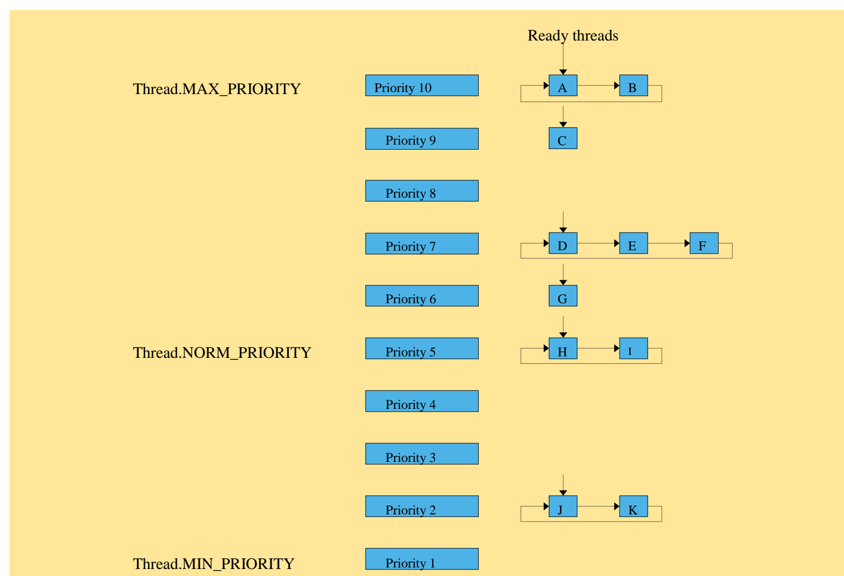
18

執行緒的排程實際上是和作業系統有關：

- 先佔先贏（preemptive scheduling）：高優先權的執行緒可持續執行，直至結束或等待，除非有更高優先權的執行緒出現。
- 時間分配（Time slicing）：CPU的使用被切成小段的時間，執行緒在使用過一單位的CPU時間後，就會進入Ready狀態，接著排程者會依優先權去挑選下一個執行者。優先權高的，有較大的機會執行。

19

Fig. 16.2 Thread priority scheduling example



20

Creating and Executing Threads

- Sleep state
  - Thread method **sleep()** called
  - Thread sleeps for a set time interval then awakens

21

```
1  // Fig. 16.3: ThreadTester.java
2  // Multiple threads printing at different intervals.
3
4  public class ThreadTester {
5
6     public static void main( String [] args )
7     {
8        // create and name each thread
9        PrintThread thread1 = new PrintThread( "thread1" );
10       PrintThread thread2 = new PrintThread( "thread2" );
11       PrintThread thread3 = new PrintThread( "thread3" );
12
13       System.err.println( "Starting threads" );
14
15       thread1.start(); // start thread1 and place it in ready state
16       thread2.start(); // start thread2 and place it in ready state
17       thread3.start(); // start thread3 and place it in ready state
18
19       System.err.println( "Threads started, main ends\n" );
20
21    } // end main
22
23 } // end class ThreadTester
24
```

Outline

ThreadTester.java

Lines 9-11

Lines 15-17

```
25  // class PrintThread controls thread execution
26  class PrintThread extends Thread {
27     private int sleepTime;
28
29     // assign name to thread by calling superclass constructor
30     public PrintThread( String name )
31     {
32        super( name );
33
34        // pick random sleep time between 0 and 5 seconds
35        sleepTime = ( int ) ( Math.random() * 5001 );
36     }
37
38     // method run is the code to be executed by new thread
39     public void run()
40     {
41        // put thread to sleep for sleepTime amount of time
42        try {
43           System.err.println(
44              getName() + " going to sleep for " + sleepTime );
45
46           Thread.sleep( sleepTime );
47        }
48
```

Outline

ThreadTester.java

Line 26

Line 35

Line 39

```
49        // if thread interrupted during sleep, print stack trace
50        catch ( InterruptedException exception ) {
51           exception.printStackTrace();
52        }
53
54        // print thread name
55        System.err.println( getName() + " done sleeping" );
56
57     } // end method run
58
59  } // end class PrintThread
```

```
Starting threads
Threads started, main ends

thread1 going to sleep for 1217
thread2 going to sleep for 3989
thread3 going to sleep for 662
thread3 done sleeping
thread1 done sleeping
thread2 done sleeping

Starting threads
thread1 going to sleep for 314
thread2 going to sleep for 1990
Threads started, main ends

thread3 going to sleep for 3016
thread1 done sleeping
thread2 done sleeping
thread3 done sleeping
```

Outline

ThreadTester.java

Thread Synchronization

- Java uses monitors for thread synchronization
- The sychronized keyword
  - Every synchronized method of an object has a monitor
  - One thread inside a synchronized method at a time
  - All other threads block until method finishes
  - Next highest priority thread runs when method finishes

25

Producer/Consumer without Synchronization

- Buffer
  - Shared memory region
- Producer thread
  - Generates data to add to buffer
  - Calls wait if consumer has not read previous message in buffer
  - Writes to empty buffer and calls notify for consumer
- Consumer thread
  - Reads data from buffer
  - Calls wait if buffer empty
- Synchronize threads to avoid corrupted data

26

```
1   // Fig. 16.4: Buffer.java
2   // Buffer interface specifies methods called by Producer and Consumer.
3
4   public interface Buffer {
5      public void set( int value );  // place value into Buffer
6      public int get();             // return value from Buffer
7   }
```

Outline

Buffer.java

```
1   // Fig. 16.5: Producer.java
2   // Producer's run method controls a thread that
3   // stores values from 1 to 4 in sharedLocation.
4
5   public class Producer extends Thread {
6      private Buffer sharedLocation; // reference to shared object
7
8      // constructor
9      public Producer( Buffer shared )
10     {
11        super( "Producer" );
12        sharedLocation = shared;
13     }
14
15     // store values from 1 to 4 in sharedLocation
16     public void run()
17     {
18       for ( int count = 1; count <= 4; count++ ) {
19
20          // sleep 0 to 3 seconds, then place value in Buffer
21          try {
22             Thread.sleep( ( int ) ( Math.random() * 3001 ) );
23             sharedLocation.set( count );
24          }
25
```

Outline

Producer.java

Line 5

Line 6

Line 16

Lines 22-23

```
26          // if sleeping thread interrupted, print stack trace
27          catch ( InterruptedException exception ) {
28             exception.printStackTrace();
29          }
30
31       } // end for
32
33       System.err.println( getName() + " done producing." +
34          "\nTerminating " + getName() + ".");
35
36    } // end method run
37
38 } // end class Producer
```

Outline

Producer.java

```
1   public class Consumer extends Thread {
6        private Buffer sharedLocation; // reference to shared object
7        public Consumer( Buffer shared )  {
11          super( "Consumer" );   sharedLocation = shared;
13       }
14
15        public void run()  {
18          int sum = 0;
20          for ( int count = 1; count <= 4; count++ ) {
22             try {
24                Thread.sleep( ( int ) ( Math.random() * 3001 ) );
25                sum += sharedLocation.get();
26             }
27             catch ( InterruptedException exception ) {
28                exception.printStackTrace();   }
32          }
33          System.err.println( getName() + " read values totaling: " + sum +
                  ".\nTerminating " + getName() + ".");
36       } // end method run
38     } // end class Consumer
```

Outline

Consumer.java

Line 5

Line 6

Line 16

Lines 24-25

```
1  // Fig. 16.7: UnsynchronizedBuffer.java
2  // UnsynchronizedBuffer represents a single shared integer.
3
4  public class UnsynchronizedBuffer implements Buffer {
5     private int buffer = -1; // shared by producer and consumer threads
6
7     // place value into buffer
8     public void set( int value )
9     {
10       System.err.println( Thread.currentThread().getName() +
11          " writes " + value );
12
13       buffer = value;
14    }
15
16    // return value from buffer
17    public int get()
18    {
19       System.err.println( Thread.currentThread().getName() +
20          " reads " + buffer );
21
22       return buffer;
23    }
24
25 } // end class UnsynchronizedBuffer
```

Outline

UnsynchronizedBuffer .java

Line 4

Line 5

Lines 8 and 13

Lines 17 and 22

```
1  // Fig. 16.8: SharedBufferTest.java
2  // SharedBufferTest creates producer and consumer threads.
3
4  public class SharedBufferTest {
5
6     public static void main( String [] args )
7     {
8        // create shared object used by threads
9        Buffer sharedLocation = new UnsynchronizedBuffer();
10
11       // create producer and consumer objects
12       Producer producer = new Producer( sharedLocation );
13       Consumer consumer = new Consumer( sharedLocation );
14
15       producer.start();  // start producer thread
16       consumer.start();  // start consumer thread
17
18    } // end main
19
20 } // end class SharedCell
```

Outline

SharedBufferTest.java

Line 9

Lines 12-13

Lines 15-16

```
Consumer reads -1
Producer writes 1
Consumer reads 1
Consumer reads 1
Consumer reads 1
Consumer read values totaling: 2.
Terminating Consumer.
Producer writes 2
Producer writes 3
Producer writes 4
Producer done producing.
Terminating Producer.
```

```
Producer writes 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer reads 4
Consumer read values totaling: 13.
Terminating Consumer.
```

Outline

SharedBufferTest.java

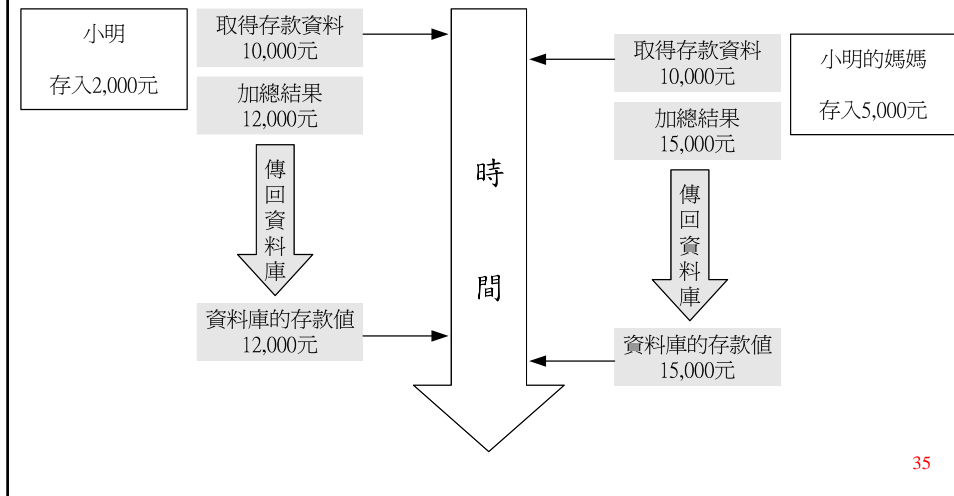
```
Producer writes 1
Consumer reads 1
Producer writes 2
Consumer reads 2
Producer writes 3
Consumer reads 3
Producer writes 4
Producer done producing.
Terminating Producer.
Consumer reads 4
Consumer read values totaling: 10.
Terminating Consumer.
```

Outline

SharedBufferTest.java

使用共同資源的多執行緒

- 多執行緒可能造成的錯誤



35

同步化方法

- 使用synchronized關鍵字修飾操作共同資源的方法，可以讓該方法不會被兩個以上的執行緒同時使用，也就是在某個時間頂多只會有一個執行緒在使用該方法。
- 擁有synchronized修飾的方法之物件，就是一個監視器（Monitor），監視器會讓物件內的synchronized方法只讓一個執行緒使用。
- 若物件中的所有方法都使用sysnchronized修飾，則在某個時間點只會有一個方法被執行。因為，**Monitor**是物件而不是方法。

36

synchronized敘述

- synchronized也可以當敘述使用

```
synchronized (物件) {
        //物件同步區
}
```

- synchronized區塊中的程式敘述還未執行完畢，該物件就不會被其它執行緒所使用。
- 當synchronized區塊在執行時，其同步化的物件的鎖（lock）是暫時被取用的。

37

synchronized敘述

- 使用synchronized修飾方法和下式同義：

```
方法型別   方法名(形式參數列) {
        synchronized(this)
        {
                //方法內敘述
        }
}
```

- synchronized的同步化對象其實是物件實體，因此synchronized不能用來修飾屬性、建構子或類別。

38

wait()及notify()

- wait()、notify()和notifyAll()方法只能使用於synchronized修飾的方法或synchronized敘述中。
- wait()方法是讓執行緒進入等待的狀態（waiting pool），同時「放棄物件的使用權」。
- wait()和sleep()兩者都可以讓執行緒進入等待狀態，不過sleep()並不會放棄物件的使用權。
- notify()會隨機叫醒waiting pool中的某個執行緒，而nitifyAll()則是叫醒waiting pool中所有的執行緒。

死結

- 在使用wait()及notify()時，必須注意避免讓執行緒進行到「等待狀態」之後出不來，如此造成執行緒永遠都在等，而且不會結束，這種情形稱為**死結**（deadlock）。
- 死結算是一種**邏輯上的錯誤**，發生死結時，並不會丟出例外，所以要格外小心避免。

Producer/Consumer with Synchronization

- Synchronize threads to ensure correct data

41

```
1  // Fig. 16.9: SynchronizedBuffer.java
2  // SynchronizedBuffer synchronizes access to a single shared integer.
3
4  public class SynchronizedBuffer implements Buffer {
5     private int buffer = -1; // shared by producer and consumer threads
6     private int occupiedBufferCount = 0; // count of occupied buffers
7
8     // place value into buffer
9     public synchronized void set( int value )
10    {
11       // for output purposes, get name of thread that called this method
12       String name = Thread.currentThread().getName();
13
14       // while there are no empty locations, place thread in waiting state
15       while ( occupiedBufferCount == 1 ) {
16
17          // output thread information and buffer information, then wait
18          try {
19             System.err.println( name + " tries to write." );
20             displayState( "Buffer full. " + name + " waits." );
21             wait();
22          }
23
24          // if waiting thread interrupted, print stack trace
25          catch ( InterruptedException exception ) {
26             exception.printStackTrace();
27          }
```

Outline

SynchronizedBuffer.java

Line 4

Line 6

Line 9

Line 12

Lines 15 and 21

```
28
29       } // end while
30
31       buffer = value; // set new buffer value
32
33       // indicate producer cannot store another value
34       // until consumer retrieves current buffer value
35       ++occupiedBufferCount;
36
37       displayState( name + " writes " + buffer );
38
39       notify(); // tell waiting thread to enter ready state
40
41    } // end method set; releases lock on SynchronizedBuffer
42
43    // return value from buffer
44    public synchronized int get()
45    {
46       // for output purposes, get name of thread that called this method
47       String name = Thread.currentThread().getName();
48
```

Outline

SynchronizedBuffer.java

Line 31

Line 35

Line 39

Line 44

Line 47

```
49       // while no data to read, place thread in waiting state
50       while ( occupiedBufferCount == 0 ) {
51
52          // output thread information and buffer information, then wait
53          try {
54             System.err.println( name + " tries to read." );
55             displayState( "Buffer empty. " + name + " waits." );
56             wait();
57          }
58
59          // if waiting thread interrupted, print stack trace
60          catch ( InterruptedException exception ) {
61             exception.printStackTrace();
62          }
63
64       } // end while
65
66       // indicate that producer can store another value
67       // because consumer just retrieved buffer value
68       --occupiedBufferCount;
69
70       displayState( name + " reads " + buffer );
71
72       notify(); // tell waiting thread to become ready to execute
73
74       return buffer;
```

Outline

SynchronizedBuffer.java

Lines 50 and 56

Line 68

Line 72

Line 74

```
75
76    } // end method get; releases lock on SynchronizedBuffer
77
78    // display current operation and buffer state
79    public void displayState( String operation )
80    {
81       StringBuffer outputLine = new StringBuffer( operation );
82       outputLine.setLength( 40 );
83       outputLine.append( buffer + "\t\t" + occupiedBufferCount );
84       System.err.println( outputLine );
85       System.err.println();
86    }
87
88 } // end class SynchronizedBuffer
```

Outline

SynchronizedBuffer.java

```
1  // Fig. 16.10: SharedBufferTest2.java
2  // SharedBufferTest2creates producer and consumer threads.
3
4  public class SharedBufferTest2 {
5
6     public static void main( String [] args )
7     {
8        // create shared object used by threads; we use a SynchronizedBuffer
9        // reference rather than a Buffer reference so we can invoke
10       // SynchronizedBuffer method displayState from main
11       SynchronizedBuffer sharedLocation = new SynchronizedBuffer();
12
13       // Display column heads for output
14       StringBuffer columnHeads = new StringBuffer( "Operation" );
15       columnHeads.setLength( 40 );
16       columnHeads.append( "Buffer\t\tOccupied Count" );
17       System.err.println( columnHeads );
18       System.err.println();
19       sharedLocation.displayState( "Initial State" );
20
21       // create producer and consumer objects
22       Producer producer = new Producer( sharedLocation );
23       Consumer consumer = new Consumer( sharedLocation );
24
```

Outline

SharedBufferTest2.java

Line 11

Line 19

Lines 22-23

```
25      producer.start(); // start producer thread
26      consumer.start(); // start consumer thread
27
28    } // end main
29
30 } // end class SharedBufferTest2
```

```
Operation                      Buffer      Occupied Count

Initial State                   -1          0

Consumer tries to read.
Buffer empty. Consumer waits.       -1          0

Producer writes 1                  1           1

Consumer reads 1                   1           0

Consumer tries to read.
Buffer empty. Consumer waits.       1           0

Producer writes 2                  2           1

Consumer reads 2                   2           0

Producer writes 3                  3           1
```

Outline

SharedBufferTest2.java

Lines 25-26

```
Consumer reads 3                   3           0

Consumer tries to read.
Buffer empty. Consumer waits.       3           0

Producer writes 4                  4           1

Consumer reads 4                   4           0
Producer done producing.
Terminating Producer.

Consumer read values totaling: 10.
Terminating Consumer.
```

```
Operation                      Buffer      Occupied Count

Initial State                   -1          0

Consumer tries to read.
Buffer empty. Consumer waits.       -1          0

Producer writes 1                  1           1

Consumer reads 1                   1           0

Producer writes 2                  2           1
```

Outline

SharedBufferTest2.java

```
Producer tries to write.
Buffer full. Producer waits.        2        1

Consumer reads 2                    2        0

Producer writes 3                   3        1

Consumer reads 3                    3        0

Producer writes 4                   4        1

Producer done producing.
Terminating Producer.
Consumer reads 4                    4        0

Consumer read values totaling: 10.
Terminating Consumer.
```

```
Operation                Buffer      Occupied Count

Initial State              -1        0

Producer writes 1           1        1

Consumer reads 1            1        0

Producer writes 2           2        1
```

Outline

SharedBufferTest2.java


```
Consumer reads 2                    2        0

Producer writes 3                   3        1

Consumer reads 3                    3        0

Producer writes 4                   4        1

Producer done producing.
Terminating Producer.
Consumer reads 4                    4        0

Consumer read values totaling: 10.
Terminating Consumer.
```

Outline

SharedBufferTest2.java

Producer/Consumer Synchronization: Circular Buffer

- Circular buffer
  - Multiple memory cells
  - Produce item if one or more empty cells
  - Consume item if one or more filled cells

51

```
1   // Fig. 16.11: RunnableOutput.java
2   // Class RunnableOutput updates JTextArea with output
3   import javax.swing.*;
4
5   public class RunnableOutput implements Runnable {
6      private JTextArea outputArea;
7      private String messageToAppend;
8
9      // initialize outputArea and message
10     public RunnableOutput( JTextArea output, String message )
11     {
12        outputArea = output;
13        messageToAppend = message;
14     }
15
16     // method called by SwingUtilities.invokeLater to update outputArea
17     public void run()
18     {
19        outputArea.append( messageToAppend );
20     }
21
22  } // end class RunnableOutput
```

Outline

RunnableOutput.java

Outline

Producer.java

```
1  // Fig. 16.12: Producer.java
2  // Producer's run method controls a thread that
3  // stores values from 11 to 20 in sharedLocation.
4  import javax.swing.*;
5
6  public class Producer extends Thread {
7     private Buffer sharedLocation;
8     private JTextArea outputArea;
9
10    // constructor
11    public Producer( Buffer shared, JTextArea output )
12    {
13       super( "Producer" );
14       sharedLocation = shared;
15       outputArea = output;
16    }
17
18    // store values from 11-20 and in sharedLocation's buffer
19    public void run()
20    {
21       for ( int count = 11; count <= 20; count ++ ) {
22
23          // sleep 0 to 3 seconds, then place value in Buffer
24          try {
25             Thread.sleep( ( int ) ( Math.random() * 3000 ) );
26             sharedLocation.set( count );
27          }
```

Outline

Producer.java

```
28
29          // if sleeping thread interrupted, print stack trace
30          catch ( InterruptedException exception ) {
31             exception.printStackTrace();
32          }
33       }
34
35       String name = getName();
36       SwingUtilities.invokeLater( new RunnableOutput( outputArea, "\n" +
37          name + " done producing.\n" + name + " terminated.\n" ) );
38
39    } // end method run
40
41 } // end class Producer
```

Outline

Consumer.java

```
1  // Fig. 16.13: Consumer.java
2  // Consumer's run method controls a thread that loops ten
3  // times and reads a value from sharedLocation each time.
4  import javax.swing.*;
5
6  public class Consumer extends Thread {
7     private Buffer sharedLocation; // reference to shared object
8     private JTextArea outputArea;
9
10    // constructor
11    public Consumer( Buffer shared, JTextArea output )
12    {
13       super( "Consumer" );
14       sharedLocation = shared;
15       outputArea = output;
16    }
17
18    // read sharedLocation's value ten times and sum the values
19    public void run()
20    {
21       int sum = 0;
22
23       for ( int count = 1; count <= 10; count++ ) {
24
```

Outline

Consumer.java

```
25          // sleep 0 to 3 seconds, read value from Buffer and add to sum
26          try {
27             Thread.sleep( ( int ) ( Math.random() * 3001 ) );
28             sum += sharedLocation.get();
29          }
30
31          // if sleeping thread interrupted, print stack trace
32          catch ( InterruptedException exception ) {
33             exception.printStackTrace();
34          }
35       }
36
37       String name = getName();
38       SwingUtilities.invokeLater( new RunnableOutput( outputArea,
39          "\nTotal " + name + " consumed: " + sum + ".\n" +
40          name + " terminated.\n " ) );
41
42    } // end method run
43
44 } // end class Consumer
```

Outline

CircularBuffer.java

```
1  // Fig. 16.14: CircularBuffer.java
2  // CircularBuffer synchronizes access to an array of shared buffers.
3  import javax.swing.*;
4  
5  public class CircularBuffer implements Buffer {
6  
7     // each array element is a buffer
8     private int buffers[] = { -1, -1, -1 };
9  
10    // occupiedBufferCount maintains count of occupied buffers
11    private int occupiedBufferCount = 0;
12 
13    // variables that maintain read and write buffer locations
14    private int readLocation = 0, writeLocation = 0;
15 
16    // reference to GUI component that displays output
17    private JTextArea outputArea;
18 
19    // constructor
20    public CircularBuffer( JTextArea output )
21    {
22       outputArea = output;
23    }
24 
```

Outline

CircularBuffer.java

```
25    // place value into buffer
26    public synchronized void set( int value )
27    {
28       // for output purposes, get name of thread that called this method
29       String name = Thread.currentThread().getName();
30 
31       // while there are no empty locations, place thread in waiting state
32       while ( occupiedBufferCount == buffers.length ) {
33 
34          // output thread information and buffer information, then wait
35          try {
36             SwingUtilities.invokeLater( new RunnableOutput( outputArea,
37                "\nAll buffers full. " + name + " waits." ) );
38             wait();
39          }
40 
41          // if waiting thread interrupted, print stack trace
42          catch ( InterruptedException exception )
43          {
44             exception.printStackTrace();
45          }
46 
47       } // end while
48 
49       // place value in writeLocation of buffers
50       buffers[ writeLocation ] = value;
51 
```

Outline

CircularBuffer.java

```
52        // update Swing GUI component with produced value
53        SwingUtilities.invokeLater( new RunnableOutput( outputArea,
54           "\n" + name + " writes " + buffers[ writeLocation ] + " " ) );
55
56        // just produced a value, so increment number of occupied buffers
57        ++occupiedBufferCount;
58
59        // update writeLocation for future write operation
60        writeLocation = ( writeLocation + 1 ) % buffers.length;
61
62        // display contents of shared buffers
63        SwingUtilities.invokeLater( new RunnableOutput(
64           outputArea, createStateOutput() ) );
65
66        notify(); // return waiting thread (if there is one) to ready state
67
68     } // end method set
69
70     // return value from buffer
71     public synchronized int get()
72     {
73        // for output purposes, get name of thread that called this method
74        String name = Thread.currentThread().getName();
75
```

Outline

CircularBuffer.java

```
76        // while no data to read, place thread in waiting state
77        while ( occupiedBufferCount == 0 ) {
78
79           // output thread information and buffer information, then wait
80           try {
81              SwingUtilities.invokeLater( new RunnableOutput( outputArea,
82                 "\nAll buffers empty. " + name + " waits." ) );
83              wait();
84           }
85
86           // if waiting thread interrupted, print stack trace
87           catch ( InterruptedException exception ) {
88              exception.printStackTrace();
89           }
90
91        } // end while
92
93        // obtain value at current readLocation
94        int readValue = buffers[ readLocation ];
95
96        // update Swing GUI component with consumed value
97        SwingUtilities.invokeLater( new RunnableOutput( outputArea,
98           "\n" + name + " reads " + readValue + " " ) );
99
100       // just consumed a value, so decrement number of occupied buffers
101       --occupiedBufferCount;
102
```

Outline

CircularBuffer.java

```
103       // update readLocation for future read operation
104       readLocation = ( readLocation + 1 ) % buffers.length;
105
106       // display contents of shared buffers
107       SwingUtilities.invokeLater( new RunnableOutput(
108          outputArea, createStateOutput() ) );
109
110       notify(); // return waiting thread (if there is one) to ready state
111
112       return readValue;
113
114    } // end method get
115
116    // create state output
117    public String createStateOutput()
118    {
119       // first line of state information
120       String output =
121          "(buffers occupied: " + occupiedBufferCount + ")\nbuffers: ";
122
123       for ( int i = 0; i < buffers.length; i++ )
124          output += " " + buffers[ i ] + "  ";
125
126       // second line of state information
127       output += "\n          ";
128
```

Outline

CircularBuffer.java

```
129       for ( int i = 0; i < buffers.length; i++ )
130          output += "---- ";
131
132       // third line of state information
133       output += "\n          ";
134
135       // append readLocation (R) and writeLocation (W)
136       // indicators below appropriate buffer locations
137       for ( int i = 0; i < buffers.length; i++ )
138
139          if ( i == writeLocation && writeLocation == readLocation )
140             output += " WR  ";
141          else if ( i == writeLocation )
142             output += " W   ";
143          else if ( i == readLocation )
144             output += "  R  ";
145          else
146             output += "     ";
147
148       output += "\n";
149
150       return output;
151
152    } // end method createStateOutput
153
154 } // end class CircularBuffer
```

Outline

CircularBufferTest.java

```
1   // Fig. 16.15: CircularBufferTest.java
2   // CircularBufferTest shows two threads manipulating a circular buffer.
3   import java.awt.*;
4   import java.awt.event.*;
5   import javax.swing.*;
6
7   // set up the producer and consumer threads and start them
8   public class CircularBufferTest extends JFrame {
9     JTextArea outputArea;
10
11    // set up GUI
12    public CircularBufferTest()
13    {
14      super( "Demonstrating Thread Synchronizaton" );
15
16      outputArea = new JTextArea( 20,30 );
17      outputArea.setFont( new Font( "Monospaced", Font.PLAIN, 12 ) );
18      getContentPane().add( new JScrollPane( outputArea ) );
19
20      setSize( 310, 500 );
21      setVisible( true );
22
23      // create shared object used by threads; we use a CircularBuffer
24      // reference rather than a Buffer reference so we can invoke
25      // CircularBuffer method createStateOutput
26      CircularBuffer sharedLocation = new CircularBuffer( outputArea );
27
```

Outline

CircularBufferTest.java

```
28      // display initial state of buffers in CircularBuffer
29      SwingUtilities.invokeLater( new RunnableOutput( outputArea,
30        sharedLocation.createStateOutput() ) );
31
32      // set up threads
33      Producer producer = new Producer( sharedLocation, outputArea );
34      Consumer consumer = new Consumer( sharedLocation, outputArea );
35
36      producer.start();  // start producer thread
37      consumer.start();  // start consumer thread
38
39    } // end constructor
40
41    public static void main ( String args[] )
42    {
43      CircularBufferTest application = new CircularBufferTest();
44      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
45    }
46
47  } // end class CirclularBufferTest
```

Outline

CircularBufferTest.java

```
(buffers occupied: 0)
buffers:  -1   -1   -1
         ---- ---- ----
          WR

All buffers empty. Consumer waits.
Producer writes 11 (buffers occupied: 1)
buffers:  11   -1   -1
         ---- ---- ----
           R    W

Consumer reads 11 (buffers occupied: 0)
buffers:  11   -1   -1
         ---- ---- ----
               WR

All buffers empty. Consumer waits.
Producer writes 12 (buffers occupied: 1)
buffers:  11   12   -1
         ---- ---- ----
                R    W

Consumer reads 12 (buffers occupied: 0)
buffers:  11   12   -1
         ---- ---- ----
                    WR

All buffers empty. Consumer waits.
Producer writes 13 (buffers occupied: 1)
buffers:  11   12   13
         ---- ---- ----
          W          R
```

Value placed in last buffer. Next value will be deposited in first buffer.

Outline

CircularBufferTest.java

```
Consumer reads 13 (buffers occupied: 0)
buffers:  11   12   13
         ---- ---- ----
          WR

Producer writes 14 (buffers occupied: 1)
buffers:  14   12   13
         ---- ---- ----
           R    W

Consumer reads 14 (buffers occupied: 0)
buffers:  14   12   13
         ---- ---- ----
               WR

Producer writes 15 (buffers occupied: 1)
buffers:  14   15   13
         ---- ---- ----
                R    W

Producer writes 16 (buffers occupied: 2)
buffers:  14   15   16
         ---- ---- ----
          W     R

Producer writes 17 (buffers occupied: 3)
buffers:  17   15   16
         ---- ---- ----
               WR
```

Circular buffer effect—the fourth value is deposited in the first buffer.

Value placed in last buffer. Next value will be deposited in first buffer.

Circular buffer effect—the seventh value is deposited in the first buffer.

Outline

CircularBufferTest.java

```
All buffers full. Producer waits.
Consumer reads 15 (buffers occupied: 2)
buffers:  17   15   16
         ---- ---- ----
               W    R

Producer writes 18 (buffers occupied: 3)
buffers:  17   18   16
         ---- ---- ----
                    WR

All buffers full. Producer waits.
Consumer reads 16 (buffers occupied: 2)
buffers:  17   18   16
         ---- ---- ----
          R         W

Producer writes 19 (buffers occupied: 3)
buffers:  17   18   19
         ---- ---- ----
          WR

All buffers full. Producer waits.
Consumer reads 17 (buffers occupied: 2)
buffers:  17   18   19
         ---- ---- ----
          W    R

Producer writes 20 (buffers occupied: 3)
buffers:  20   18   19
         ---- ---- ----
               WR
```

Value placed in last buffer. Next value will be deposited in first buffer.

Circular buffer effect—the tenth value is deposited in the first buffer.

Outline

CircularBufferTest.java

```
Producer done producing.
Producer terminated.

Consumer reads 18 (buffers occupied: 2)
buffers:  20   18   19
         ---- ---- ----
               W    R

Consumer reads 19 (buffers occupied: 1)
buffers:  20   18   19
         ---- ---- ----
          R    W

Consumer reads 20 (buffers occupied: 0)
buffers:  20   18   19
         ---- ---- ----
               WR

Total Consumer consumed: 155.
Consumer terminated.
```

Daemon Threads

- Run for benefit of other threads
  - Do not prevent program from terminating
  - Garbage collector is a daemon thread
- Set daemon thread with method setDaemon

69

Runnable **Interface**

- A class cannot extend more than one class
- Implement Runnable for multithreading support

70

```
1   // Fig. 16.16: RandomCharacters.java
2   // Class RandomCharacters demonstrates the Runnable interface
3   import java.awt.*;
4   import java.awt.event.*;
5   import javax.swing.*;
6
7   public class RandomCharacters extends JApplet implements ActionListener {
8      private String alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
9      private final static int SIZE = 3;
10     private JLabel outputs[];
11     private JCheckBox checkboxes[];
12     private Thread threads[];
13     private boolean suspended[];
14
15     // set up GUI and arrays
16     public void init()
17     {
18        outputs = new JLabel[ SIZE ];
19        checkboxes = new JCheckBox[ SIZE ];
20        threads = new Thread[ SIZE ];
21        suspended = new boolean[ SIZE ];
22
23        Container container = getContentPane();
24        container.setLayout( new GridLayout( SIZE, 2, 5, 5 ) );
25
```

Outline

RandomCharacters.java

```
26        // create GUI components, register listeners and attach
27        // components to content pane
28        for ( int count = 0; count < SIZE; count++ ) {
29           outputs[ count ] = new JLabel();
30           outputs[ count ].setBackground( Color.GREEN );
31           outputs[ count ].setOpaque( true );
32           container.add( outputs[ count ] );
33
34           checkboxes[ count ] = new JCheckBox( "Suspended" );
35           checkboxes[ count ].addActionListener( this );
36           container.add( checkboxes[ count ] );
37        }
38
39     } // end method init
40
41     // create and start threads each time start is called (i.e., after
42     // init and when user revists Web page containing this applet)
43     public void start()
44     {
45        for ( int count = 0; count < threads.length; count++ ) {
46
47           // create Thread; initialize object that implements Runnable
48           threads[ count ] =
49              new Thread( new RunnableObject(), "Thread " + ( count + 1 ) );
50
51           threads[ count ].start(); // begin executing Thread
52        }
```

Outline

RandomCharacters.java

Line 43

Lines 48-49

Line 51

```
53     }
54
55     // determine thread location in threads array
56     private int getIndex( Thread current )
57     {
58        for ( int count = 0; count < threads.length; count++ )
59           if ( current == threads[ count ] )
60              return count;
61
62        return -1;
63     }
64
65     // called when user switches Web pages; stops all threads
66     public synchronized void stop()
67     {
68        // set references to null to terminate each thread's run method
69        for ( int count = 0; count < threads.length; count++ )
70           threads[ count ] = null;
71
72        notifyAll(); // notify all waiting threads, so they can terminate
73     }
74
75     // handle button events
76     public synchronized void actionPerformed( ActionEvent event )
77     {
```

Outline

RandomCharacters.java

Line 66

Line 70

Line 72

```
78        for ( int count = 0; count < checkboxes.length; count++ ) {
79
80           if ( event.getSource() == checkboxes[ count ] ) {
81              suspended[ count ] = !suspended[ count ];
82
83              // change label color on suspend/resume
84              outputs[ count ].setBackground(
85                 suspended[ count ] ? Color.RED : Color.GREEN );
86
87              // if thread resumed, make sure it starts executing
88              if ( !suspended[ count ] )
89                 notifyAll();
90
91              return;
92           }
93        }
94
95     } // end method actionPerformed
96
97     // private inner class that implements Runnable to control threads
98     private class RunnableObject implements Runnable {
99
100       // place random characters in GUI, variables currentThread and
101       // index are final so can be used in an anonymous inner class
102       public void run()
103       {
```

Outline

RandomCharacters.java

Line 81

Line 89

Line 98

Line 102

```
104      // get reference to executing thread
105      final Thread currentThread = Thread.currentThread();
106
107      // determine thread's position in array
108      final int index = getIndex( currentThread );
109
110      // loop condition determines when thread should stop; loop
111      // terminates when reference threads[ index ] becomes null
112      while ( threads[ index ] == currentThread ) {
113
114         // sleep from 0 to 1 second
115         try {
116            Thread.sleep( ( int ) ( Math.random() * 1000 ) );
117
118            // determine whether thread should suspend execution;
119            // synchronize on RandomCharacters applet object
120            synchronized( RandomCharacters.this ) {
121
122               while ( suspended[ index ] &&
123                  threads[ index ] == currentThread ) {
124
125                  // temporarily suspend thread execution
126                  RandomCharacters.this.wait();
127               }
128            } // end synchronized statement
```

Outline

RandomCharacters.java

Line 112

Line 120

Line 126

```
129
130         } // end try
131
132         // if thread interrupted during wait/sleep, print stack trace
133         catch ( InterruptedException exception ) {
134            exception.printStackTrace();
135         }
136
137         // display character on corresponding JLabel
138         SwingUtilities.invokeLater(
139            new Runnable() {
140
141               // pick random character and display it
142               public void run()
143               {
144                  char displayChar =
145                     alphabet.charAt( ( int ) ( Math.random() * 26 ) );
146
147                  outputs[ index ].setText(
148                     currentThread.getName()  + ": " + displayChar );
149               }
150
151            } // end inner class
152
153         ); // end call to SwingUtilities.invokeLater
```

Outline

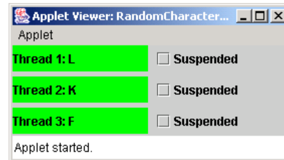
RandomCharacters.java

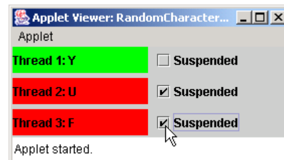
Line 139

```
154
155         } // end while
156
157         System.err.println( currentThread.getName() + " terminating" );
158
159      } // end method run
160
161   } // end private inner class RunnableObject
162
163 } // end class RandomCharacters
```

Outline

RandomCharacters.java





放棄該次CPU的使用機會

- 為了避免CPU被獨佔，可以使用**yield()**方法讓某個執行緒進入Ready狀態，而暫時先讓出CPU。
- 呼叫高優先權執行緒的yield()方法，可以讓低優先權執行緒有較多的機會使用到CPU，以避免低優先權執行緒處於挨餓狀態（starvation）。

78

等待狀態

- 執行sleep()方法：暫停執行緒一段時間。
- 執行suspend()方法：沒有時間限制地暫停執行緒，可呼叫resume()方法回到Ready狀態。
- 呼叫join()方法：呼叫目標執行緒的join()方法，將等到目標執行緒結束之後才會繼續。
- Input/Output blocked：執行緒進行輸入輸出時，會因為輸入輸出的速度較慢而暫時進入Waiting。
- 呼叫同步方法時，未取得物件的lock。
- 執行wait()方法：放棄物件的使用權並進入等待狀態，可使用notify()方法喚醒執行緒。

79

問題

- 撰寫一多執行緒程式，其中一個執行緒每隔200毫秒列出一偶數，另一執行緒每隔300毫秒列出一3的倍數，印到100就結束
- 撰寫一賽馬程式，有五匹馬，馬匹每隔10毫秒會隨機前進一段小於1公尺的距離。主程式則是每隔10毫秒列出每一匹馬的目前距離（列出整數），當有一匹馬跑到終點時，則程式結束

80