Threads

Some examples are implemented in Java to execute as an applet.

Threads Overview

- A thread is an execution.
- All programs have at least one thread of execution control.
- C# supports additional multiple, light-weight threads of execution.
- Running two separate programs at the same time is an example of a heavy-weight threads, each is a single program in execution, the internal *environments* of other heavy-weight threads are isolated from one another.
- A heavy-weight program thread can have multiple light-weight threads, each being switched to execute for a time then suspended while another thread executes.
- The prime distinction between heavy and light weight threads are that light threads share a portion of a common environment where heavy-weight threads are isolated.
- When light-weight threads are switched to allow one to execute there is less individual thread information to store and switch since much is common to all.

Execute one heavy thread.

```
using System;
public class one
{
   public static void Main( )
   {
        Console.Write ("heavy one");
   }
}
```

Execute heavy and light thread.

```
using System;
using System.Threading;
public class one {
   public static void Main()  {
      new two();
      Console.Write("heavy one");
   }
}
```

C# Thread Comparison

```
class two {
   public two() {
      Thread newTwo=
           new Thread(new ThreadStart(run));
      newTwo.Start();
   }

   public void run()  {
       Console.Write("light two");
    }
}
```

C# thread support

The key advantages of C# thread support are:

1. communication between threads is relatively simple
2. inexpensive in execution since light-weight, not necessary to save the entire program state when switching between threads
3. part of language, allows a program to handle multiple, concurrent operations in an operating system platform independent fashion.
4. C# on *preemptive* operating systems ensure that all threads will get to execute occasionally by *preempting* thread execution (stopping the currently executing thread and running another) but this does not ensure that threads execute in any particular order.

C# Thread Execution

5 creates new *two* object.

11 creates new *Thread* object, defines **run** as handler.

12 calls handler **`run`** and returns immediately**.**

6 and 15 execute independently, i.e. asynchronously.

16 **`run`** completes execution but does not return.

Possible execution sequences:

5, 11, 12, 6, 15
5, 11, 12, 15, 6

```
1.  using System;
2.  using System.Threading;
3.  public class one {
4.      public static void Main()  {
5.          new two();
6.          Console.Write("heavy one");
7.      }
8.  }
```

```
9.  class two {
10.    public two() {
11.      Thread newTwo=
             new Thread(new ThreadStart(run));
12.       newTwo.Start();
13.     }

14.     public void run()  {
15.         Console.Write("light two");
16.      }
17. }
```

Java Single Heavy-Weight Thread

- We'll use a bouncing ball (number) to illustrate differences in multi-threaded execution.
- Our thread example uses a *single* heavy weight thread to bounce a *single* ball off the sides of the display window.
- The **Ball** class implements the ball bouncing, since a single thread is used, only one ball can be bounced at a time. The Ball class implements two public methods:
  - *Ball()* - constructs a Ball object.
  - *move()* - moves a Ball a predefined distance in the current direction and changes direction to a complementary angle if a wall is encountered.
- Video - SingleThread.swf

Only one thread is running. The key part of the program is:

```
public void run() {
 for (int i = 1; i <= 1000; i++) {
  move();
  try { Thread.sleep(10); } catch(Exception e) {}
 }
}
```

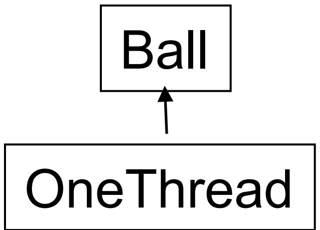
invoked once as a normal function call from onClick( ) function by clicking the Toss button. The button click:

1. executes the **run()** method
2. the ball is *move*d (*move* is inherited from the *Ball* class),
3. the iteration in **run()** *moves* the ball
4. **Thread.sleep(10);** puts the main, heavy weight thread to sleep for at least 10 milliseconds; giving up the CPU allowing other threads to run.
5. With 1000 iterations (each sleeping 10 milliseconds), the ball *move*s for at least 10 seconds.

```
1. public class SingleThread extends UserInterface {
2.   public SingleThread() { addButton("Toss"); }

3.   public void onClick(char c){
4.    OneThread ot = new OneThread(this);
5.     ot.run();
6.   }
7. }

8. class OneThread extends Ball {
9.   public OneThread(UserInterface ui) {
10.     super(ui);
11.   }

12.   public void run() {
13.    for (int i = 1; i <= 1000; i++) {
14.      move();
15.      try { Thread.sleep(10); } catch(Exception e) {}
16.    }
17.   }
18.}
```

Exercise 1

1. List in order line(s) executed when *Toss* button is clicked.
2. How many threads execute?

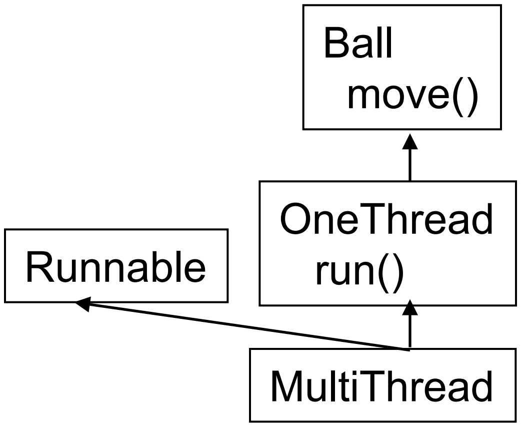


Java Multiple Light-Weight Threads

- With only minor changes to the previous example, multiple light-weight threads can support many simultaneously bouncing balls. Key changes are:

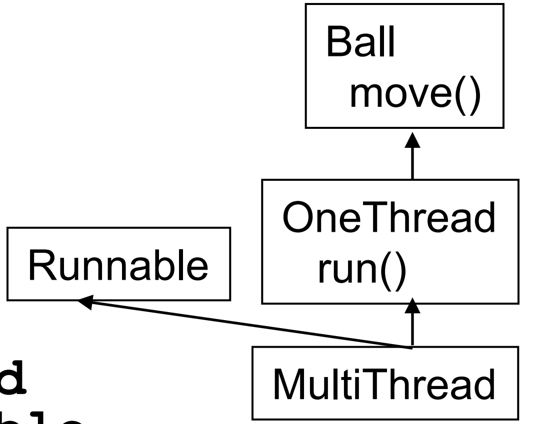
1. Implement the **Runnable** class for light weight threads.
2. Extend **OneThread** class to start a new thread for each ball constructed when Toss clicked.

```
class MultiThread extends OneThread implements Runnable {
  public MultiThread(UserInterface ui) {
      super(ui);
      new Thread(this).start();
  }
}
```



- Video - MultipleThreads.swf

Java Runnable interface



- MultiThread subclass of OneThread and Runnable:

```
class MultiThread  extends OneThread
                   implements Runnable
```

- **`implements Runnable`** contract to implement method definition:
  `public void run()`

- Thread constructed by:
  `new Thread(this)`

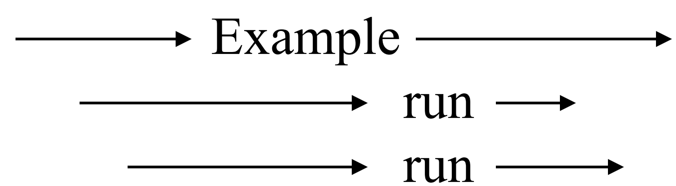
- **`run()`** invoked by **`start()`**

- **`start();`** starts a thread and returns immediately

- **`new Thread(this).start();`** constructs and starts a new thread.

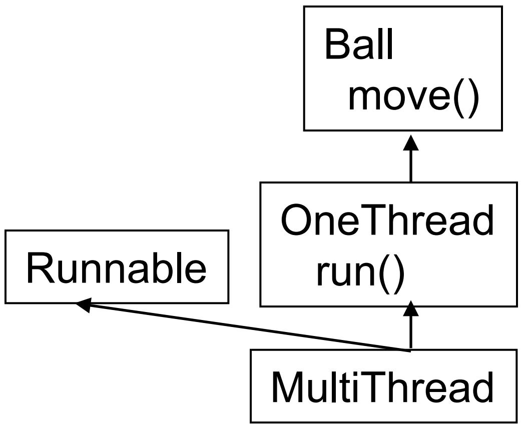
- **`Thread.sleep(10);`** causes a thread to give up the CPU.

```
1. public class Example extends UserInterface {
2.    public Example() {
3.      addButton("Toss");
4.    }
5.    public void onClick(char c) {
6.      MultiThread mt = new MultiThread(this);
7.    }
8. }
9. class MultiThread extends OneThread
                             implements Runnable{
10.   public MultiThread(UserInterface ui) {
11.     super(ui);
12.     new Thread(this).start();
13.   }
14.}
```



1. Where is the *run()* method?
2. List the lines executed when *Toss* is clicked the first time. See next slide.
3. What happens to a thread at the end of the *run( )* method?
4. What does *new Thread(this).start( )* do, since you can press *Toss* repeatedly and another ball starts bouncing?

```java
public class Example extends UserInterface {
 public Example() {
   addButton("Toss");
 }
 public void onClick(char c) {
   MultiThread mt = new MultiThread(this);
 }
}
class MultiThread extends OneThread implements Runnable{
 public MultiThread(UserInterface ui) {
   super(ui);
   new Thread(this).start();
 }
}
class OneThread extends Ball {
 public OneThread(UserInterface ui) {
   super(ui);
 }
 public void run() {
  for (int i = 1; i <= 1000; i++) {
   move();
   try { Thread.sleep(10); } catch(Exception e) {}
  }
 }
```



C# Execution Non-deterministic

```
using System;
using System.Threading;
class ST {
   public static void Main() {
      new Simple(1);
      new Simple(2);
      new Simple(3);
      new Simple(4);
   }
}
class Simple {
  int n;
  public Simple(int n) {
     this.n = n;
    (new Thread(new ThreadStart(run))).Start();
  }
  public void run() {
      Thread.Sleep(10);
      Console.WriteLine( n );
      Console.Out.Flush();
  }
}
```

```
4 Executions
>C# ST
2
1
3
4
>C# ST
2
1
3
4
> C# ST
1
3
2
4
> C# ST
1
2
4
3
```

C# Array of Threads

```
using System;
using System.Threading;
class AT {
   public static void Main() {
      Simple [] sa = new Simple[10];
      for (int i=0; i<10; i++) sa[ i ] = new Simple( i );
   }
}
class Simple {
  int n;
  public Simple(int n) {
    this.n = n;
    (new Thread(new ThreadStart(run))).Start();
  }
  public void run() {
      Thread.Sleep(10);
      Console.WriteLine( n );
      Console.Out.Flush();
  }
}
```

A thread is an object that can be stored in any data structure.

3 Executions

4	0	2
3	1	3
2	2	4
1	3	5
0	4	6
9	5	7
8	6	8
7	7	9
6	8	0
5	9	1

Java Array of Greedy Threads

- Threads naturally run asynchronously to one another with no cooperation.
- Only one thread executes at a time on the CPU
- Race conditions can develop as multiple threads attempt to occupy the CPU exclusively
- Example of many threads racing to complete
- Only significant changes are:
  1. 10 threads started without artificial delay.
  2. No artificial delay introduced through the **`Thread.sleep(10);`**
- Video - ArrayOfGreedyThreads.swf

```
1.  public class ArrayOfGreedyThreads extends UserInterface {
2.    public ArrayOfGreedyThreads() {  addButton("Toss");    }
3.    public void onClick(char c) {
4.     GreedyThread gt[] = new GreedyThread[10];
5.     for(int i=0;i<10;i++)
6.          gt[i] = new GreedyThread(this);
7.    }
8.  }

9.  class GreedyThread extends Greedy
10.                   implements Runnable {
11.   public GreedyThread(UserInterface ui) {
12.    super(ui);
13.    new Thread(this).start();
14.   }
15. }

16. class Greedy extends Ball {
17.   public Greedy (UserInterface ui) {
18.         super(ui); }
19.   public void run() {
20.    for (int i = 1; i <= 1000; i++) { move(); }
21.   }
22. }
```

Exercise 3

What sequence of line(s) are executed when *Toss* is clicked (note for statements in onClick() and run() methods)?

C# Thread Race

Exercise 4 – Give sample output that **cannot** occur.

```
class AT {
  public static void Main() {
    new Simple("A");    new Simple("B");
    for (int i = 1; i <= 5; i++)
        Console.WriteLine("main " + i);
  }
}
class Simple {
  String name;
  public Simple(string name) {
   this.name = name;
   (new Thread(new ThreadStart(run))).Start();
  }
  public void run() {
    for (int i = 1; i <= 5; i++) {
        Console.WriteLine(name + " " + i);
        Console.Out.Flush();  Thread.Sleep(10);
    }
  }
}
```

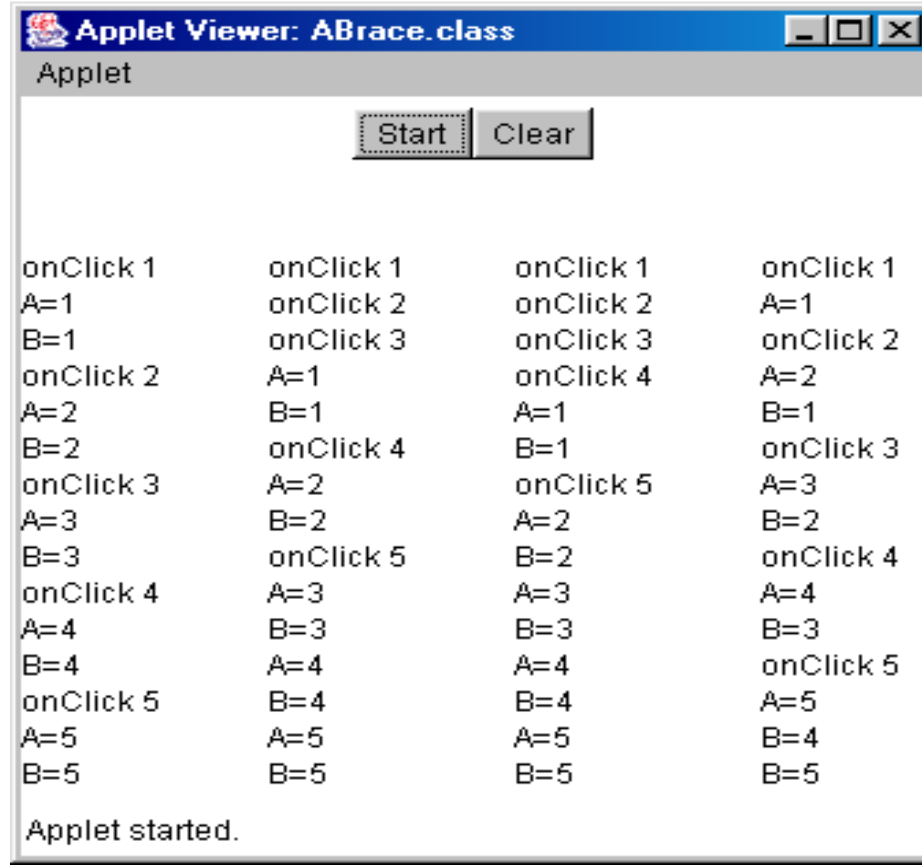
Sequential Execution	**Threaded Execution**
A 1	A 1
A 2	B 1
A 3	A 2
A 4	main 1
A 5	main 2
B 1	main 3
B 2	main 4
B 3	main 5
B 4	A 3
B 5	B 2
main 1	B 3
main 2	A 4
main 3	B 4
main 4	A 5
main 5	B 5

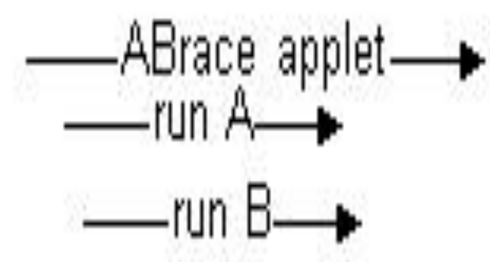
Thread Race Conditions

- Threads compete for execution time
- Scheduled for execution non-deterministically (i.e. order of execution is not predetermined).
- Unless thread execution completes, the thread is suspended and another thread given the CPU
- Threads can execute independently
- Race conditions occur when two or more threads interfere with other thread's results.
- Example illustrates how thread execution is arbitrary suspended by the run time system.
- Executing the example produces differences in printed results.
- Video – ABrace.swf

Java Race Output

- There are 3 threads
  1. ABrace in onClick()
  2. A
  3. B
- Note the far right column
- The threads a racing each other to execute





```
public class ABrace extends UserInterface {
  public ABrace() { addButton("Start");  addButton("Clear");   }
  public void init() { Display.initialize(this); }
  public void onClick(char c) {
    Race A, B;
    switch(c) {
       case 'S' : Display.heading(30);
                  A = new Race("A");
                  B = new Race("B");
                  for (int i=1; i<=5; i++)
                    Display.println("onClick "+i);
                  break;
       case 'C' : Display.clear();
     }
  }
}

class Race implements Runnable {
   String name;
   public Race( String name )
   {  this.name = name;
      new Thread(this).start();
   }
   public void run()
   {  for (int i=1; i<=5; i++)
        Display.println(name +"="+i);
   }
}
```

Exercise 4

1. Execute the program several times.
2. How many threads are executing? Explain how you know.
3. Can the results of 5 executions be identical? Explain.

Synchronization

- Thread execution can be:
  - independent of other threads (i.e. *parallel* or *asynchronous* execution)
  - dependent (i.e. *serialized* or *synchronous* execution) where one thread executes to the exclusion of the other threads
  - Object access is controlled in C# using the *lock* statement which limits object access to one thread at a time.
- The synchronization mechanism used by C# is termed a *monitor* (versus a semaphore, task, or other mechanism)
- A *monitor* allows only one thread to have access to an object at a time.
- The keyword *lock* defines a statement or method where one thread at a time has exclusive access to the object.

```
using System;
using System.Threading;
class RT {
   public static void Main() {
       new Racer(1); new Racer(2);
       new Racer(3); new Racer(4);
  }
}
class Racer {
  int n;
  public Racer(int n) {
    this.n = n;
    (new Thread(new ThreadStart(run))).Start();
  }
  public void run() {
    Console.Write("["+n);    Console.Out.Flush();
    Thread.Sleep(10);
    Console.WriteLine(n+"]");  Console.Out.Flush();
  }
}
```

C# Non-cooperating non-serialized

```
>C# RT
[1[2[3[41]
3]
2 ]
4 ]
```

```
>C# RT
[1[2[3[44]
3]
1]
2]
```

Exercise 5a – Valid?

```
[11]
[22]
[33]
[44]
```

```
using System;      using System.Threading;
class ST {
  public static void Main() {
   new Racer(1); new Racer(2); new Racer(3);
  }
}
class Racer {
  int n;
  static string common = "common";
  public Racer(int n) {
    this.n = n;
    (new Thread(new ThreadStart(run))).Start();
  }
  public void run() {
     lock (common)  {          // Serialization block
       Console.Write("[" + n); Console.Out.Flush();
       Thread.Sleep(10);
       Console.WriteLine(n + "]"); Console.Out.Flush();
     }
  }
}
```

C# Non-cooperating Serialized

```
>C# ST
C:\d
[11]
[22]
[33]

>C# ST
[11]
[33]
[22]
```

Exercise 5b – Valid?

```
[1[2[33]
1]
2]
```

lock

- In the example above the object *common* is accessible to the threads 1, 2, and 3.
- In the *run* method, access to *common* is *serialized* so that only one thread can execute the code block:

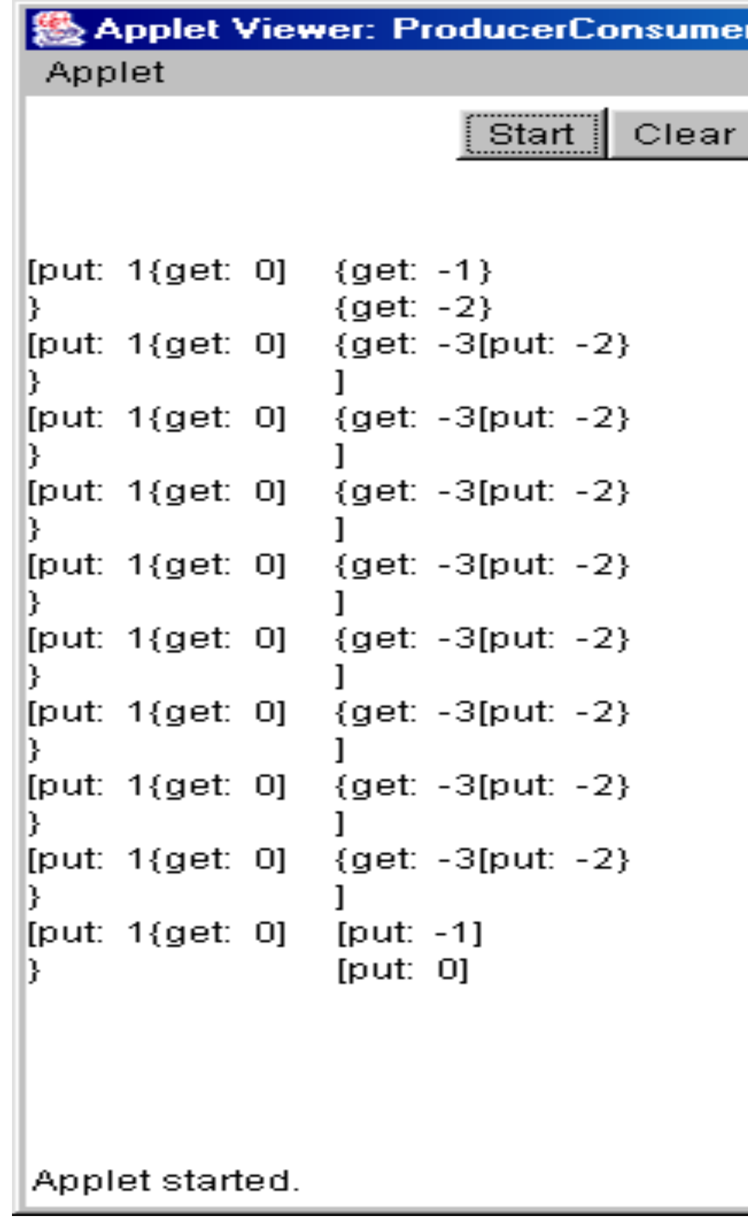
```
class Racer {
   static String common="common";
   int n;

   public void run() {
      lock( common ) {

         Console.Write("[" + n);      Console.Out.Flush();
         Thread.Sleep(10);
         Console.WriteLine(n + "]"); Console.Out.Flush();

      }
   }
```

Non-Cooperating Threads

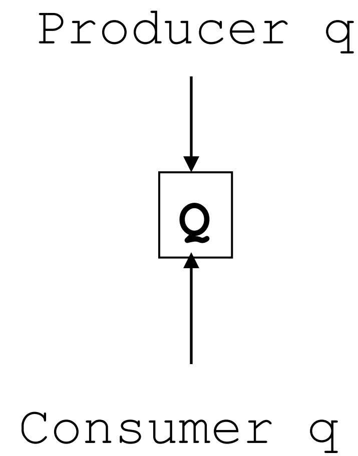
- Example of three independent threads:
  - the main heavy-weight
  - *Producer* that puts onto a common queue
  - *Consumer* that gets from the common queue
- *Consumer* thread may *get* from the queue before the *Producer* thread *puts* onto the queue
- Any thread execution may be preempted at any time
- No guarantee a *put* or *get* finished before another thread runs
- Most obvious when the printed output is intermixed as in:
  - *[put: 1 {get:0}]* rather than:
  - *[put: 1]*
    {get: 0}

Non-Cooperating Threads

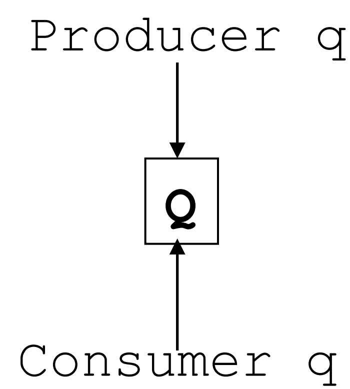
- *put* and *get* output intermixed, one not completed before other starts
- Need *serialized* access to queue object
- Even worse, *get* executed when queue is empty
- Need to force *get* execution to wait when queue is empty



```
public class ProducerConsumer {
  public static void Main() {
    Q q = new Q( );
    new Consumer(q);
    new Producer(q);
  }
}

class Q {
  int n = 0;

  public void put() {
    n = n + 1;
    Display.print("[put:  " + n);  Display.print("]\n");
  }

  public void get() {
    n = n - 1;
    Display.print("{get:  " + n);  Display.print("}\n");
  }
}
```



```
class Producer {
  Q q;
  public Producer( Q q ) {
    this.q = q;
    (new Thread(new ThreadStart(run))).Start();
  }
  public void run() {
    for (int i=1; i<=10; i++) q.put();  // Produce
  }
}

class Consumer {
  Q q;
  public Consumer( Q q ) {
    this.q = q;
    (new Thread(new ThreadStart(run))).Start();
  }
  public void run() {
    for (int i=1; i<=10; i++) q.get();   // Consume
  }
}
```



Exercise 6

1. The *Producer* and *Consumer* are both threads, each having a *run( )* method. Starting in the *run( )* of each, list the sequence of lines executed that would print:

   [put: 1{get:0]
   }

2. Again, starting in the *run( )* of each thread, list the sequence that would print:

   {get:-1 [put: 0]
   }

```
1.   using System;  using System.Threading;
2.   public class ProducerConsumer {
3.     public static void Main() {
4.       Q q = new Q( );
5.       new Consumer(q);
6.       new Producer(q);
7.     }
8.   }
9.   static class Display {
10.      public static void print(string s) {
11.          Console.Write(s); Console.Out.Flush();
12.          Thread.Sleep(10);
13.      }
14.  }
15.  class Q  {
16.   int n = 0;
17.   public void put()    {
18.     n = n + 1;
19.     Display.print("[put:" + n); Display.print("]\n");
20.   }
21.   public void get() {
22.     n = n - 1;
23.     Display.print("{get:" + n); Display.print("}\n");
24.   }
25.  }
```

```
26.  class Producer {
27.   Q q;
28.   public Producer( Q q ) {
29.    this.q = q;
30.    (new Thread(
        new ThreadStart(run))).Start();
31.    }
32.  void run() {
33.    for (int i=1; i<=10; i++)
34.       q.put();  // Produce
35.    }
36.  }
37.  class Consumer {
38.   Q q;
39.   public Consumer( Q q ) {
40.    this.q = q;
41.    (new Thread(
        new ThreadStart(run))).Start();
42.   }
43.   void run() {
44.     for (int i=1; i<=10; i++)
45.       q.get();   // Consume
46.   }
47.  }
48.
```

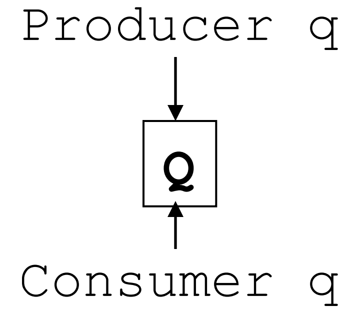
- Threads interact when simultaneously accessing common resource.
- C# supports a control mechanism called a *monitor* that allows only one thread at a time to execute a *serialized* method on an object.
- By adding *lock(this) { }* within a method, a thread entering {} has exclusive access to that object.
- When *{ }* completed, other threads have access to the object.
- For a *queue* object, the *put* and *get* method ensures either completes before another thread enters either, for the common object.
- With two queue objects, one thread could have access to one queue object while another thread accessed the other queue object.
- This does not ensure that something has been put into the queue before a thread attempts to get it, that is another problem.

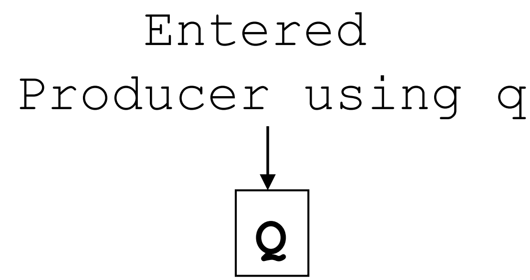
Monitor Behavior

- Monitor controls access to a *serialized* object.
- Monitor allows only one thread to access object; all other threads are blocked.
- When thread exits *serialized* area other threads can access *serialized* object.
- Blocked threads are automatically allowed to attempt to access *serialized* object again.
- Threads can still access object *unserialized*, without a *lock*.
- Code is not serialized, objects are.
- No conflict when threads access different objects.

Monitor

```
class Q {
int n = 0;
  public void put( ) {
    lock(this) {
     n = n + 1;
     Display.print("[put:" + n);
     Display.print("]\n");
    }
  }
  public void get( ) {
   lock(this) {
    n = n - 1;
    Display.print("{get:" + n);
    Display.print("}\n");
   }
  }
}
```





Consumer blocked for q

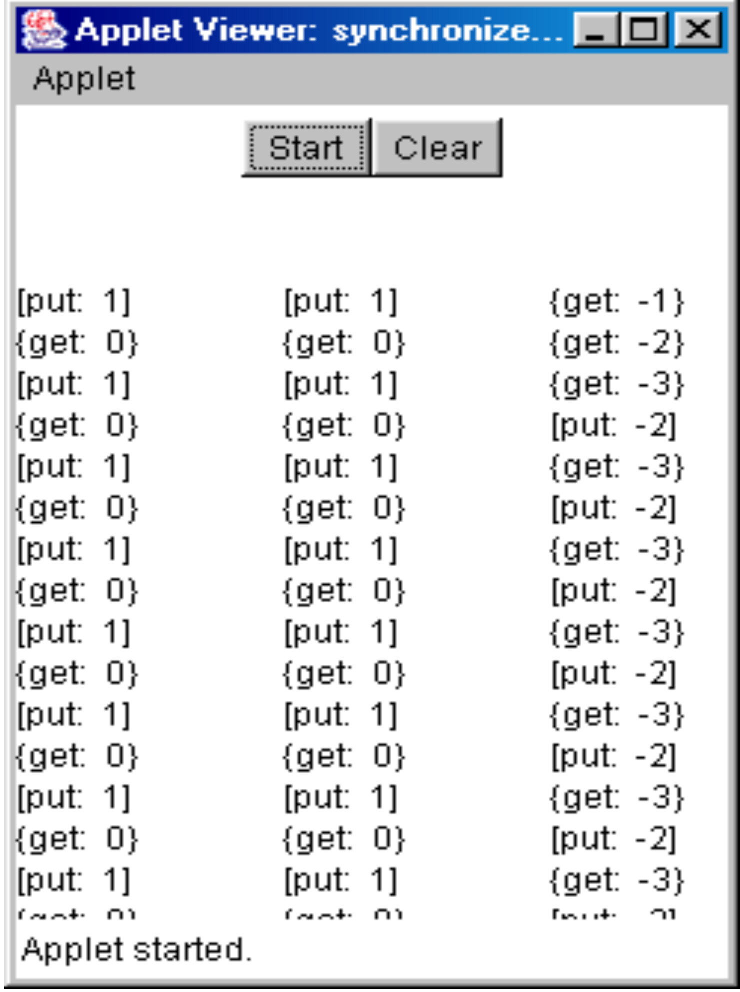
```
new Producer(q);
   :
 q.put();
```

```
new Consumer(q);
   :
q.get();
```

Consumer blocked when common Q object in use by Producer

Synchronized Queue

```
1.  class Q {
2.  int n = 0;
3.    public void put( ) {
4.     lock(this) {
5.      n = n + 1;
6.      Display.print("[put:" + n);
7.      Display.print("]\n");
8.     }
9.    }
10.   public void get( ) {
11.    lock(this) {
12.     n = n - 1;
13.     Display.print("{get:" + n);
14.     Display.print("}\n");
15.    }
16.   }
17. }
```



Exercise 7

1. Only one thread executes any serialized section on a single object. Assume *Consumer* thread is executing Line 12 when *Producer* thread executes Line 4; list the lines then executed in the above code.
2. Is serializing only the *get* method sufficient to prevent intermixed output?

Monitor:Wait, Pulse, PulseAll

- `Wait` suspends thread until `Pulse` executed by another thread on the same object
- `Pulse` sends signal to <u>one</u> waiting thread
- Which thread notified is non-deterministic
- `PulseAll` sends signal to <u>all</u> waiting threads
- Only one notified thread can execute any *serialized* method on an object at a time
- A notified thread continues from point where `Wait` executed
- Thread enters a dead state when `run` completes
- The monitor controls access to a *serialized* object
- No conflict when threads access different objects

Monitor Behavior

- Monitor controls access to *serialized* object.
- Difference between threads *blocked* because monitor busy with another thread and threads that explicitly called **`Wait`**

  - When *serialized* method completes, blocked threads automatically re-attempt object access

  - Threads that called **`Wait`** can only proceed by another thread calling **`Pulse`** or **`PulseAll`** on the same object

Monitor

```
1. class Q {
2.   int n = 0;
3.   public void put( ) {
4.    lock(this) {
5.     n = n + 1;
6.     Display.print("[put:"+ n);
7.     Display.print("]\n");
8.     Monitor.Pulse(this);
9.    }
10.  }
11.  public void get( ) {
12.   lock(this) {
13.    while ( n==0 ) Monitor.Wait(this);
14.    n = n - 1;
15.    Display.print("{get:  " + n);
16.    Display.println("}");
17.   }
18.  }
19.}
   Producer p = new Producer(q); … q.put();
   Consumer c = new Consumer(q); … q.get();
```

12. Consumer: *lock* q.
13. Consumer: n==0, Wait(*this*) release *q*
4. Producer: *lock* q.
8. Producer: Pulse(*this*) to thread waiting on *q*.
13. Consumer: receives Pulse(*this*) on *q*, obtains *lock*. n==1 then continues.
14. Consumer: …

Producer using q

Q

Consumer waits for q when n==0

Cooperating Producer Consumer

- **`Wait()`** suspends thread until **`Pulse()`** executed by another thread
- **`Pulse( )`** sends signal to waiting thread
- Continues from point where **`Wait()`** executed.

Exercise 8a

1. **`while ( n==0 )`** is needed. Why?
2. What happens without line 8?
3. Without line 17?

```
1. class Q {
2.    int n = 0;
3.    public void put( ) {
4.     lock(this) {
5.      n = n + 1;
6.      Display.print("[put:"+ n);
7.      Display.print("]\n");
8.      Monitor.Pulse(this);
9.     }
10.   }
11.   public void get( ) {
12.    lock(this) {
13.     while ( n==0 )
              Monitor.Wait(this);
14.     n = n - 1;
15.     Display.print("{get:  " + n);
16.     Display.println("}");
17.     Monitor.Pulse(this);
18.    }
19.   }
20.}
```

Cooperating Producer Consumer Busy Wait

Exercise 8a

3. Line 17 not needed. Why?

Exercise 8b

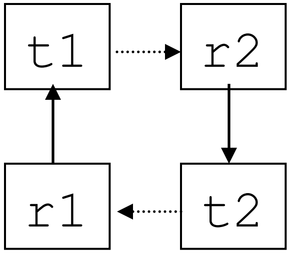
4. This seems to prevent consuming before producing. How?

5. What is the problem?

```
1. class Q {
2.   int n = 0;
3.   public void put( ) {
4.     n = n + 1;
5.     Display.print("[put:  " + n);
6.     Display.print("]\n");
7.
8.   }
9.   public void get( ) {
10.    while ( n==0 );
11.
12.
13.
14.    n = n - 1;
15.    Display.print("{get:  " + n);
16.    Display.print("}\n");
17.    Monitor.Pulse(this);
18.  }
19.}
```

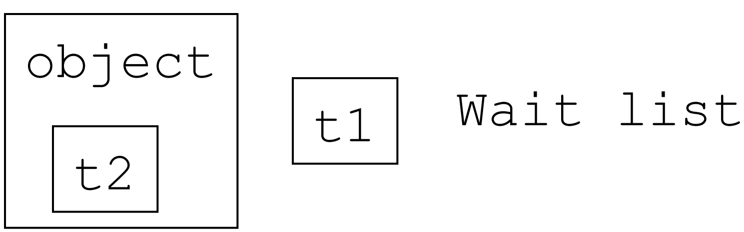
Deadlock

- Deadlock occurs when one thread cannot complete.
- One example is deadly embrace (or circular wait) where two threads each hold a resource required by the other.
  - t1 and t2 threads require both resources r1 and r2 to complete.
  - t1 thread holds r1 resource; t2 thread holds r2 resource.
  - t1 and t2 both deadlocked.



Starvation

- t1, t2 – Attempt *lock* on common object.
- t1 claims lock and executes Wait();
- t2 claims lock, Pulse() not executed.
- t1 never completes.



Multi-thread Summary

- Thread creation - Use **`Thread`** class

```
(new Thread(new ThreadStart(run))).Start();
```

- Serialization – Monitor automatically limits execution of code on a shared object to one thread using *lock*
- Basic thread control
  - Wait – Places a thread on wait list for object.
  - Pulse – Releases an arbitrary thread from wait list for object.
  - PulseAll – Releases all threads from wait list for object.
  - Sleep – Suspends thread execution for a minimum specified time.

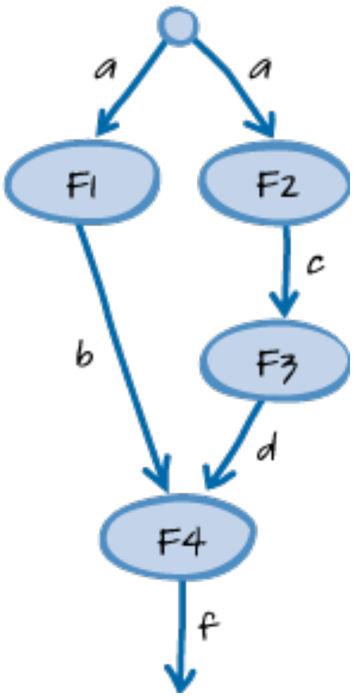
Task Parallel Library

The Task Parallel Library (TPL) is based on the concept of the task. The term task parallelism refers to one or more independent tasks running concurrently. A task represents an asynchronous operation, and in some ways it resembles the creation of a new thread or ThreadPool work item, but at a higher level of abstraction. Tasks provide two primary benefits:

1. More efficient and more scalable use of system resources.
2. Behind the scenes, tasks are queued to the ThreadPool, which has been enhanced with algorithms (like hill-climbing) that determine and adjust to the number of threads that maximizes throughput. This makes tasks relatively lightweight, so can create many of them to enable fine-grained parallelism.

You can find more detail in Parallel Programming with Microsoft®.NET, Colin Campbell, Ralph Johnson, Ade Miller, Stephen Toub (an earlier draft is online @ http://parallelpatterns.codeplex.com/).

Futures https://msdn.microsoft.com/en-us/library/ff963556.aspx

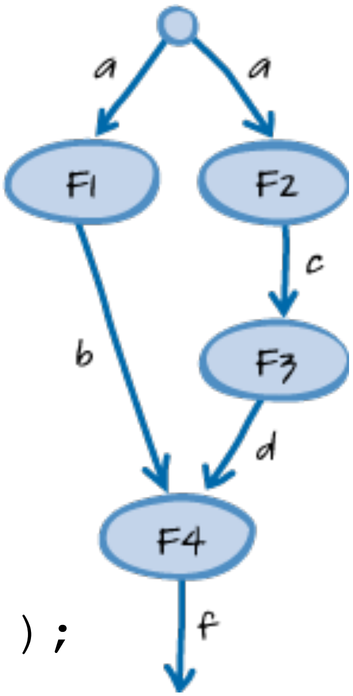


- F1 can execute in parallel (asynchronously) with F2 and F3.
- F2 and F3 must execute sequentially (synchronously).
- F4 cannot execute until both F1 and F3 complete.

```
int a = 1;
int b = F1(a);
int c = F2(a);
int d = F3(c);
int f = F4(b + d);
```

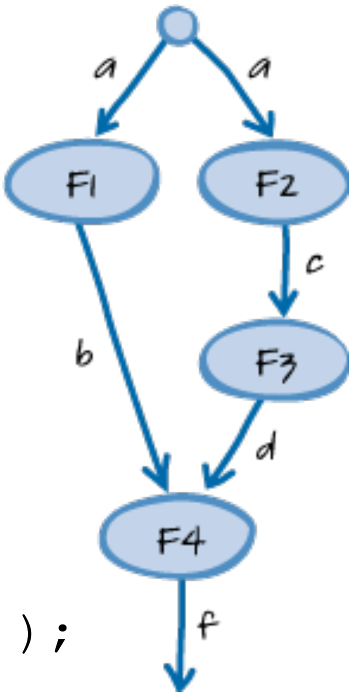
- Future is a result determined at some later time.
- F4 must block until both results b and d are computed.
- Functions (without side-effects) are more easily used in futures.

Futures https://msdn.microsoft.com/en-us/library/ff963556.aspx




```
using System;
using System.Threading.Tasks;
namespace Future{
  class Program    {
   static void Main(string[] args)        {
      int a = 1;
      Task<int> futureB =
         Task.Factory.StartNew<int>(() => F1(a));
      int c = F2(a);
      int d = F3(c);
      int f = F4(futureB.Result + d);   // block for
      System.Console.Write(f);          // futureB.Result
   }
   static int F1(int a) { return a+1; }
   static int F2(int a) { return a+1; }
   static int F3(int a) { return a+1; }
   static int F4(int b) { return b+1; }
  }
}
```

Exercise 10 – Output?



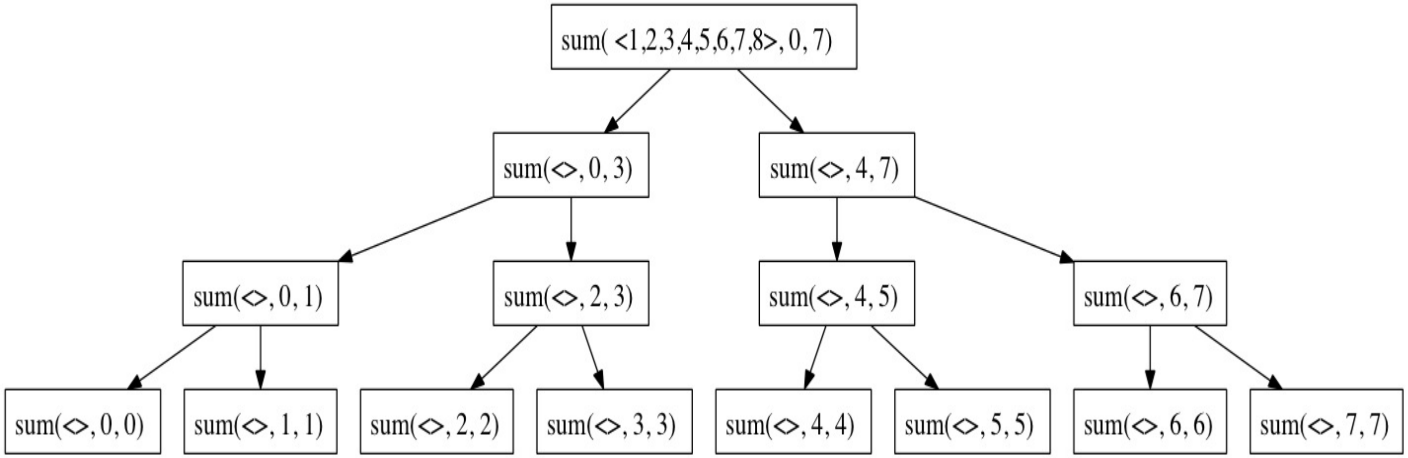

```
using System;
using System.Threading.Tasks;
namespace Future{
  class Program    {
   static void Main(string[] args)       {
      int a = 1;
      Task<int> futureB =
         Task.Factory.StartNew<int>(() => F1(a));
      int c = F2(a);
      int d = F3(c);
      int f = F4(futureB.Result + d);    // Block for
      System.Console.Write(f);           // futureB.Result
   }
   static int F1(int a) { return a+1; }
   static int F2(int a) { return a+1; }
   static int F3(int a) { return a+1; }
   static int F4(int b) { return b+1; }
 }
}
```

```
using System;
using System.Threading.Tasks;

public class Program
{
   public static void Main() {
      Task task1 = new Task(( ) => printMessage("1") );
      Task task2 = new Task(( ) => printMessage("2") );

      task1.Start();
      task2.Start();
      Console.WriteLine("Main complete.");
      Console.ReadLine();
   }
   private static void printMessage(string s)
   {
      Console.WriteLine(s + " executing");
   }
}
```

```
3 tasks, 6 possible
orders, P(3,3):
 M12
 M21
 1M2
 12M
 2M1
 21M

Output
  Main complete.
  1 executing
  2 executing
Or
  1 executing
  Main complete.
  2 executing
Or
  1 executing
  2 executing
  Main complete.
Or …
```

```
using System;
using System.Threading.Tasks;
public class Program
{
    private static int Sum(int n)
    {
        int sum = 0;
        for (; n > 0; n--)
            sum += n;
        return sum;
    }

    public static void Main()
    {
        Task<int> t = new Task<int>(( ) => Sum( 1000 ) );
        t.Start();
        Console.WriteLine("The sum is: " + t.Result);          // Block, Result is int value
    }
}
```

Halve size each Sum call, O(lg n) when infinite tasks Process each half by new task asynchronously.




```
using System;
using System.Threading.Tasks;
public class Program
{
   public static void Main()
   {
      int [] A = {1,2,3,4,5,6,7,8};
      Console.WriteLine(" Sum : "
         + Sum(A, 0, 7));
   }
```

```
static int Sum(int [] A, int l, int r) {
      if( l >= r) return A[l];            // Array size 1
      int m=(int)Math.Floor((double)((l+r)/2));

      Task<int> tl = new Task<int>(( ) => Sum(A, l, m));
      Task<int> tr = new Task<int>(( ) => Sum(A, m+1, r));
      tl.Start();
      tr.Start();
      return tl.Result + tr.Result;   // Wait on Results
   }
}
```

Parallel.For loops

Data parallelism refers to scenarios in which the same operation is performed concurrently (that is, in parallel) on elements in a source collection or array. Data parallelism with imperative syntax is supported by several overloads of the *For* and *ForEach* methods in the System.Threading.Tasks.Parallel class.

In data parallel operations, the source collection is partitioned so that multiple threads can operate on different segments concurrently.

Below has 3 potentially concurrent calls to *Sum*, limited by number of available cores.

```
Parallel.For(0, 3, i => {
    Sum( A[i] );
});
```

The first two parameters of *Parallel.For* method specify the beginning and terminating iteration values: 0, 1, 2.

for loops

```
using System;
using System.Threading;
using System.Threading.Tasks;
class Example
{
    static void Main()
    {
        for(int i=0; i<3; i++)
        {
            for(char j=‘a’; j<‘d’; j++)
            {
                Thread.Sleep(17);
                Console.WriteLine(i + " " + (char) j);
            };
        };
    }
}
```

Only one possible execution order for i or j.

abc 012

Output

```
i j
0 a
0 b
0 c
1 a
1 b
1 c
2 a
2 b
2 c
```

Parallel.For loops

```
using System;
using System.Threading;
using System.Threading.Tasks;
class Example
{
    static void Main()
    {
        for(int i=0; i<3; i++)
        {
            Parallel.For('a', 'd', j =>
            {
                Thread.Sleep(17);
                Console.WriteLine(i + " " + (char) j);
            });
        };
    }
}
```

6 possible parallel execution orders for *j*, and one *i* order.

abc	012
acb	012
bac	012
bca	012
cab	012
cba	012

Output

i	j
0	a
0	c
0	b
1	b
1	c
1	a
2	a
2	b
2	c

Exercise 11a
Possible?

i	j
1	b
0	a
1	a
0	b
0	c
1	c
2	a
2	b
2	c

Parallel.For loops

```
using System;
using System.Threading;
using System.Threading.Tasks;
class Example
{
    static void Main()
    {
        Parallel.For(0, 3, i =>
        {
            for(char j='a’; j<'d’; j++)
            {
                Thread.Sleep(17);
                Console.WriteLine(i + " " + (char) j);
            };
        });
    }
}
```

6 possible parallel execution orders for *i*, one order for *j*.

abc	012
abc	021
abc	102
abc	120
abc	210
abc	201

Output

i	j
1	a
0	a
1	b
0	b
0	c
1	c
2	a
2	b
2	c

Exercise 11b Possible?

i	j
1	b
0	a
1	a
0	b
0	c
1	c
2	a
2	b
2	c

Parallel.For loops

```
using System;
using System.Threading;
using System.Threading.Tasks;
class Example
{
    static void Main()
    {
        Parallel.For(0, 3, i =>
        {
            Parallel.For('a', 'd', j =>
            {
                Thread.Sleep(17);
                Console.WriteLine(i + " " + (char) j);
            });
        });
    }
}
```

36 possible parallel execution orders, 6 *j* orders for each *i* value, and 6 *i* orders.

abc	012
acb	021
bac	102
bca	120
cab	210
cba	201

Exercise 11c

Output	Possible?
i j	i j
1 a	1 b
0 a	0 a
1 b	1 a
0 b	0 b
0 c	0 c
1 c	1 c
2 a	2 a
2 b	2 b
2 c	2 c

for loops

Compute each y[i] separately

Each instance i=0,1,2 has one sequential execution order of:

y[i] = y[i] + A[**j**][i] * x[**j**];

i =2:

y[2] = y[2] + A[**0**][2] * x[**0**];

y[2] = y[2] + A[**1**][2] * x[**1**];

y[2] = y[2] + A[**2**][2] * x[**2**];

Each *i* executes j=0,1,2 order

```
using System;
using System.Threading.Tasks;
class ParallelMatMul
{
    static void Main()
    {
        double[][] A = {
            new double[] { 1, 1, 1 },
            new double[] { 2, 2, 2 },
            new double[] { 3, 3, 3 } };
        double[] x = { 1, 2, 3 };
        double[] y = new double[x.Length];
        for (int i = 0; i < y.Length; i++)
            y[i] = 0.0;
        for(int i=0, i<3, i++)
        {
            for(int j=0;  j<3;  j++)
            {
                y[i] = y[i] + A[j][i] * x[j];
            });
        }
        for (int i = 0; i < y.Length; i++)
            Console.WriteLine(y[i]);
    }
}
```

Parallel.For loops

Each parallel instance i=0,1,2 has one sequential execution order of:

y[i] = y[i] + A[**j**][i] * x[**j**];

i =2:

y[2] = y[2] + A[**0**][2] * x[**0**];
y[2] = y[2] + A[**1**][2] * x[**1**];
y[2] = y[2] + A[**2**][2] * x[**2**];

Each *i* executes j=0,1,2 order

```
using System;
using System.Threading.Tasks;
class ParallelMatMul
{
    static void Main()
    {
        double[][] A = {
            new double[] { 1, 1, 1 },
            new double[] { 2, 2, 2 },
            new double[] { 3, 3, 3 } };
         double[] x = { 1, 2, 3 };
         double[] y = new double[x.Length];
         for (int i = 0; i < y.Length; i++)
               y[i] = 0.0;
```

```
         Parallel.For(0, 3, i =>
         {
            for(int j=0;  j<3;  j++)
            {
               y[i] = y[i] + A[j][i] * x[j];
            });
         }
         for (int i = 0; i < y.Length; i++)
            Console.WriteLine(y[i]);
    }
}
```

Parallel.For Race Conditions

Each sequential i=0,1,2 has 6 possible parallel execution orders:

y[i] = y[i] + A[**j**][i] * x[**j**];

i =2:

y[2] = y[2] + A[**1**][2] * x[**1**];
y[2] = y[2] + A[**2**][2] * x[**2**];
y[2] = y[2] + A[**0**][2] * x[**0**];

Execute j=0,1,2 in any order.

```
using System;
using System.Threading.Tasks;
class ParallelMatMul
{
    static void Main()
    {
        double[][] A = {
            new double[] { 1, 1, 1 },
            new double[] { 2, 2, 2 },
            new double[] { 3, 3, 3 } };
         double[] x = { 1, 2, 3 };
         double[] y = new double[x.Length];
         for (int i = 0; i < y.Length; i++)
                y[i] = 0.0;
        for (int i = 0; i < 3; i++)
        {
            Parallel.For(0, 3, j =>
            {
                y[i] = y[i] + A[j][i] * x[j];
            });
        }
        for (int i = 0; i < y.Length; i++)
            Console.WriteLine(y[i]);
    }
}
```

Parallel.For Race Conditions

36 possible parallel execution orders, 6 for each *i* value:

y[i] = y[i] + A[**j**][i] * x[**j**];

i =2:

y[2] = y[2] + A[**1**][2] * x[**1**];
y[2] = y[2] + A[**0**][2] * x[**0**];
y[2] = y[2] + A[**2**][2] * x[**2**];

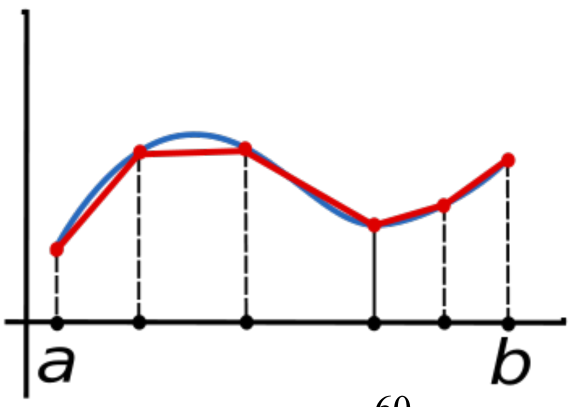
```
using System;
using System.Threading.Tasks;
class ParallelMatVec
{
    static void Main()
    {
        double[][] A = {
            new double[] { 1, 1, 1 },
            new double[] { 2, 2, 2 },
            new double[] { 3, 3, 3 } };
        double[] x = { 1, 2, 3 };
        double[] y = new double[x.Length];
        for (int i = 0; i < y.Length; i++)
            y[i] = 0.0;
        Parallel.For(0, 3, i =>
        {
            Parallel.For(0, 3, j =>
            {
                y[i] = y[i] + A[j][i] * x[j];
            });
        });
        for (int i = 0; i < y.Length; i++)
            Console.WriteLine(y[i]);
    }
}
```

Parallel vs Sequential For

```
using System.Threading.Tasks;   using System.Diagnostics;  using System;
class Program {
    static void Main(string[] args) {
        Console.Write("Cores: ");
        int cores = Convert.ToInt32(Console.ReadLine());
        var watch = Stopwatch.StartNew();
        double sum = 0;
        double [] result = new double[cores];

        Parallel.For(0, cores, (i) =>
        {
            result[i] = Integrate(x => Math.Sin(x), 10000000,
                                (i * Math.PI) / cores, ((i + 1) * Math.PI) / cores);
            sum += result[i];
            Console.WriteLine("Area {0} : Result {1} : Sum {2}", i, result[i], sum);
        });
        Console.WriteLine("Parallel Elapsed time: {0} ", watch.ElapsedMilliseconds);
```

Parallel vs Sequential For

```
watch = Stopwatch.StartNew();
sum = 0;
for (int i = 0; i < cores; i++) {
    result[i] = Integrate(x => Math.Sin(x), 10000000,
                        (i * Math.PI) / cores, ((i + 1) * Math.PI) / cores);
    sum += result[i];
    Console.WriteLine("Area {0} : {1} : {2}", i, result[i], sum);
  };
  Console.WriteLine("Serial Elapsed time: {0} ", watch.ElapsedMilliseconds);
}
// trapezoidal method
public static double Integrate(Func<double, double> f, int n, double a, double b) {
  double deltax = (b-a) / n;
  double result = f(a)/2 + f(a+deltax*n)/2;
  for (int i = 1; i < n; i++)  result += f(deltax*i+a);
   return result*deltax;
}
}
```



Parallel vs Sequential For

Finding area under sine wave
from 0 to pi using trapezoidal method .

Parallel.For uses 4 cores in example.

```
Cores: 4

Area 0 : Result 0.292893218813453 : Sum 0.292893218813453
Area 3 : Result 0.292893218813473 : Sum 0.585786437626926
Area 1 : Result 0.707106781186533 : Sum 1.29289321881346
Area 2 : Result 0.707106781186422 : Sum 1.99999999999988
Parallel Elapsed time: 691

Area 0 : Result 0.292893218813453 : Sum 0.292893218813453
Area 1 : Result 0.707106781186533 : Sum 0.999999999999986
Area 2 : Result 0.707106781186422 : Sum 1.70710678118641
Area 3 : Result 0.292893218813473 : Sum 1.99999999999988
Serial Elapsed time: 3066
```

Delegates and Callbacks

A delegate is a type that references a method. Once a delegate is assigned a method, it behaves exactly like that method. The delegate method can be used like any other method, with parameters and a return value.

```
public delegate void aDelegate(int result);

class Example {
   static void Main() {
      Process p = new Process(myDelegate);
      p.start();
   }
   public static void myDelegate(int result) {
      System.Console.WriteLine(result);
   }
}
class Process{
   aDelegate d ;
   public Process(aDelegate newD) {
      d = newD;
   }
   public void start() {
      for (int i = 0; i < 10; i++) d(i);  // Callback
   }
}
```

myDelegate has signature of *aDelegate*.

myDelegate procedure reference is passed to *Process* constructor and stored in attribute *d*.

myDelegate is called indirectly by *d(i)*.

```
Exercise 12
Single thread.
Output?
```

Delegates and Callbacks

```
using System.Threading;

public delegate void aDelegate(string result);
class Example {
    static void Main( ) {
        Process p = new Process(myDelegate);
        for (int i = 0; i < 10; i++)
                myDelegate(i+"");       // Direct call
    }
    public static void myDelegate(string result) {
                System.Console.WriteLine( result );
    }
}
class Process {
    aDelegate d ;
    public Process(aDelegate newD) {
        d = newD;
        Thread newProcess =
              new Thread(new ThreadStart(run));
        newProcess.Start();
    }
    public void run()  {                      // Indirect call
        for (char i = 'A'; i < 'I'; i++)    d(i+"");
    }
}
```

```
A
0
1
2
B
C
D
E
F
G
H
3
4
5
6
7
8
9
```

Callbacks common in threaded applications. Typically used by a long-running child thread to signal processing completed the main thread.

Note *myDelegate* is executed on two different threads.

Not a problem here but serious when child thread attempts to modify the state maintained by another, such as updating an interactive display. Need to synchronize access to state.

Events

(http://msdn.microsoft.com/en-us/library/aa645739(v=vs.71).aspx)

An event in C# is a way for a class to provide notifications to clients of that class when some interesting thing happens to an object. The most familiar use for events is in graphical user interfaces; typically, the classes that represent controls in the interface have events that are notified when the user does something to the control (for example, click a button).

Events, however, need not be used only for graphical interfaces. Events provide a generally useful way for objects to signal state changes that may be useful to clients of that object. Events are an important building block for creating classes that can be reused in a large number of different programs.

Events are declared using delegates. A delegate object encapsulates a method so that it can be called anonymously. An event is a way for a class to allow clients to give it delegates to methods that should be called when the event occurs.

When the event occurs, the delegate(s) given to it by its clients are invoked.

Events

```
public class Publisher {

   public event TickHandler Tick;          // same event/delegate
   public delegate void TickHandler(string s);

   public void Start()    {
      while (true)
         if (Tick != null) Tick( “Testing”);          // Call registered
   }
}

public class Subscriber {
   private string message;

   public Subscriber(Publisher p, string message)   {
    p.Tick += new Publisher.TickHandler(HeardIt);  // Register
    this.message = message;
   }

   private void HeardIt(string s) {
     System.Console.WriteLine(s + “ “ + message);
   }
}
```

```
class Test {
   static void Main( )  {
      Publisher p = new Publisher();

      new Subscriber(p, “Two”);
      new Subscriber(p, “One”);
      new Subscriber(p, “Three”);
      p.Start();
   }
}
```

Subscriber registers *HeardIt* as a TickHandler with Publisher, called by Publisher when generating Tick *event*.

Tick Subscriber objects

Two	One	Three

Exercise 13

Trace *Tick*(“Testing”)

Output?