Threads & Networking

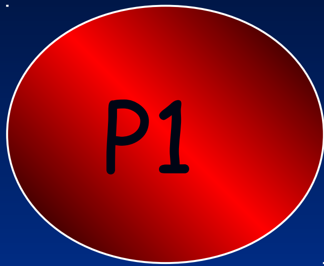
C# offers facilities for multi threading and network programming

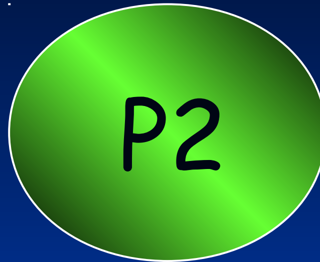
an application roughly corresponds to a process, handled by the OS

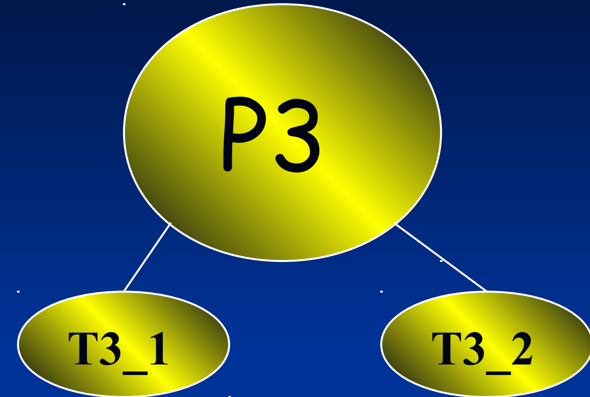
time sharing simulates multi tasking

inside an application : several execution threads

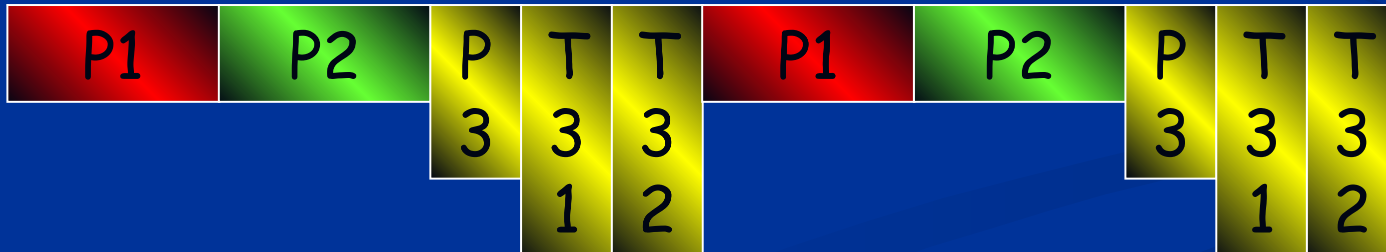
Multitasking







time



Multithreading

thread = 'light' process

several threads belonging to a single application share the same memory space : efficient communication

the main process is a thread : the main thread

easy creation of a thread associated to a method :

the method is ran independantly of the program

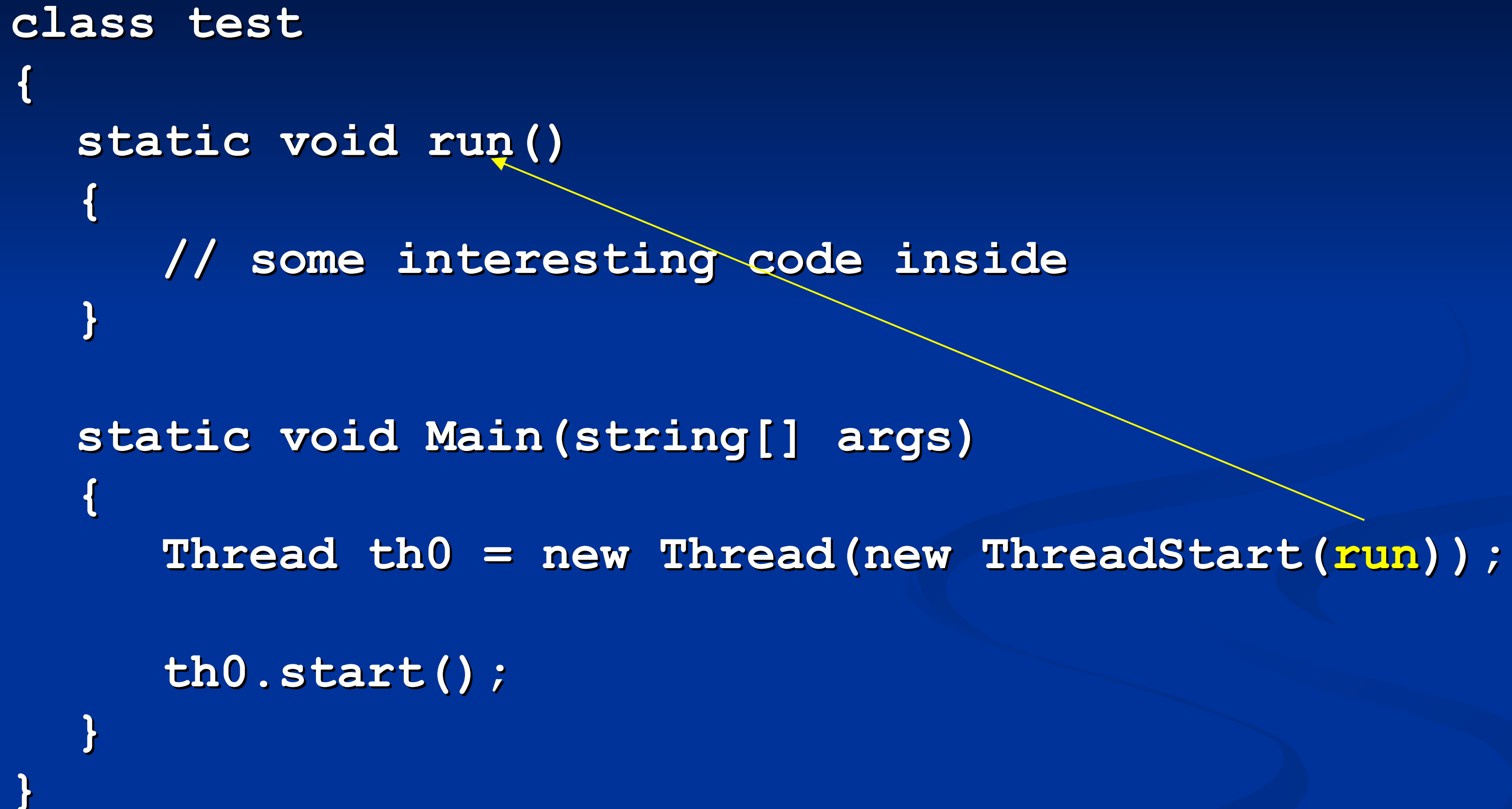
Multithreading

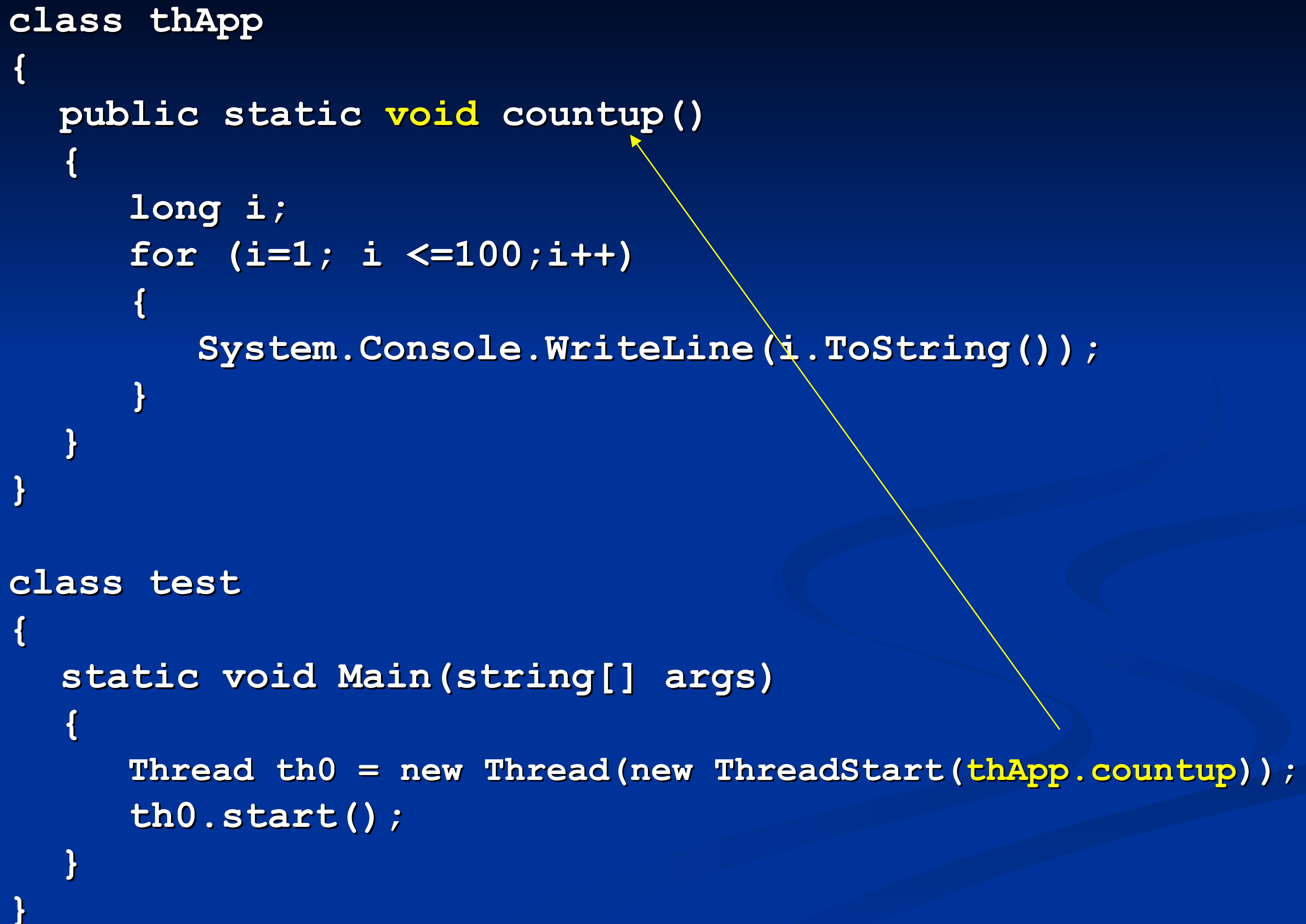
the method is run 'out of sequence' (asynchronously) and sometimes needs synchronization with other threads.

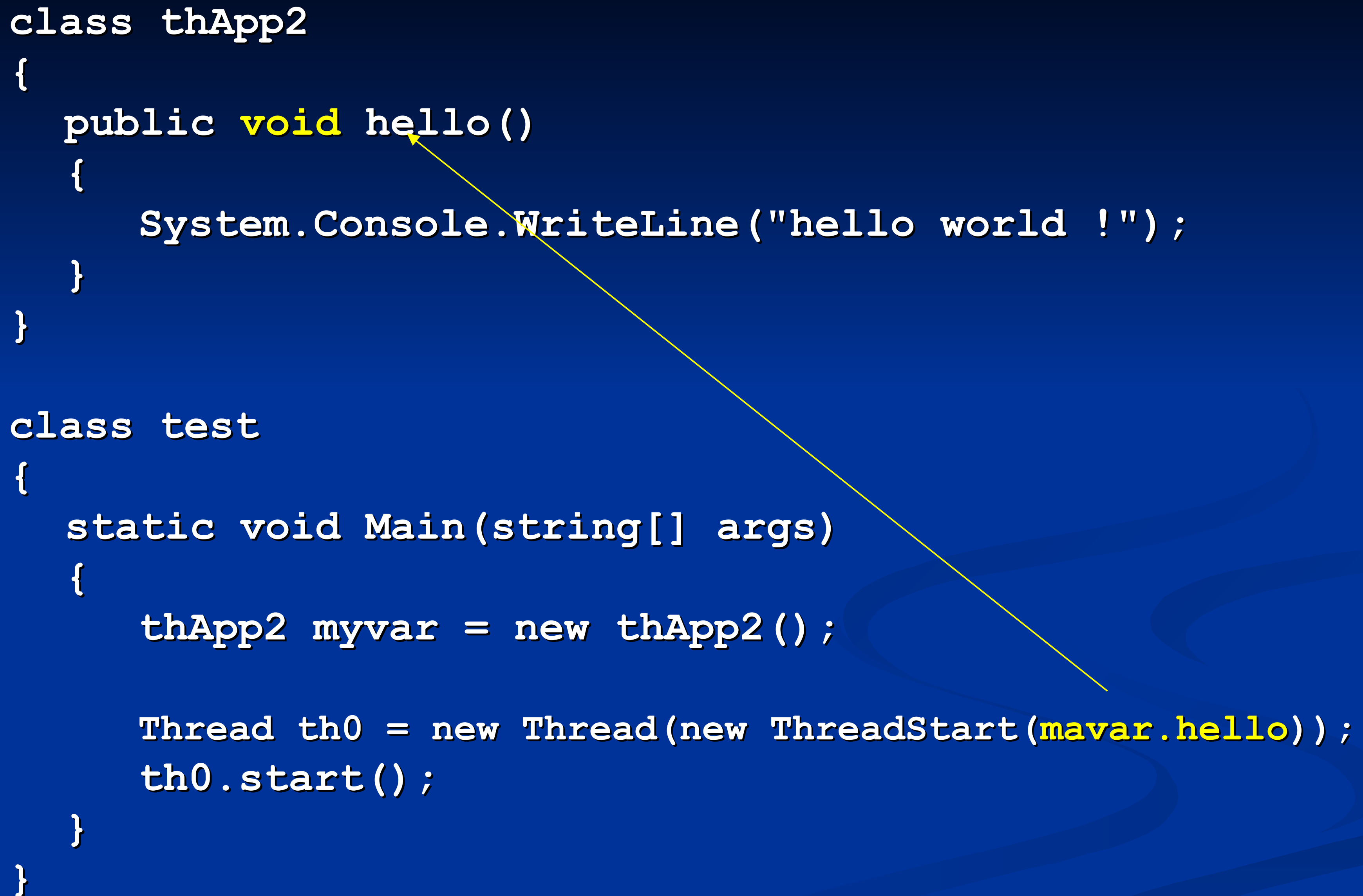
Thread class implements _Thread interface

public sealed class Thread : _Thread
{…}

Creating a thread : example




class thApp
{
public static void countup()
{
long i;
for (i=1; i <=100;i++)
{
System.Console.WriteLine(i.ToString());
}
}
}
class test
{
static void Main(string[] args)
{
Thread th0 = new Thread(new ThreadStart(thApp.countup));
th0.start();
}
}


class thApp2
{
public void hello()
{
System.Console.WriteLine("hello world !");
}
}
class test
{
static void Main(string[] args)
{
thApp2 myvar = new thApp2();
Thread th0 = new Thread(new ThreadStart(mavar.hello));
th0.start();
}
}

Parameterized Thread

```
Foo parameter = // get parameter value
Thread thread = new Thread(new ParameterizedThreadStart(DoMethod));
thread.Start(parameter); // overloaded method

// signature of function should not be changed ! (part of a delegate)
private void DoMethod(object obj)
{
    Foo parameter = (Foo)obj;
    // ...
}
```

Parmeterized Thread

- The best way to do it is to use your own class that contains state

- Type safe compared to the previous example that passes an object
  - This could raise an exception if the object is not an instance of the correct type !

Parameterized Thread

```
public class ThreadWithState
{
    // State information used in the task.
    private string boilerplate;
    private int value;

    // The constructor obtains the state information.
    public ThreadWithState(string text, int number)
    {
        boilerplate = text;
        value = number;
    }

    // The thread procedure performs the task, such as formatting
    // and printing a document.
    public void ThreadProc()
    {
        Console.WriteLine(boilerplate, value);
    }
}
```

Parameterized Thread

```
public static void Main()
    {
        // Supply the state information required by the task.
        ThreadWithState tws = new ThreadWithState(
            "This report displays the number {0}.", 42);

        // Create a thread to execute the task, and then
        // start the thread.
        Thread t = new Thread(new ThreadStart(tws.ThreadProc));
        t.Start();
        Console.WriteLine("Main thread does some work, then waits.");
        t.Join();
        Console.WriteLine(
            "Independent task has completed; main thread ends.");
    }
```

Destroying a Thread

- Use « Abort » function
- Stops the thread and the CLR raises throws a ThreadAbortException in the target thread
- The Abort method does not cause the thread to abort immediately
  - the target thread can catch the ThreadAbortException and execute arbitrary amounts of code in a finally block

Synchronization

```
class test
{
   static void Main(string[] args)
   {
   Thread th0 = new Thread(new threadStart(thApp.countup));
   th0.start();

   thApp2 myvar = new thApp2();

   th0 = new Thread(new ThreadStart(myvar.hello));
   th0.start();

   // don't forget the pause
   }
}
```

Synchronization

executing the program 3 times gives the following result :

```
1
2
3
…
22
hello world !
23
24
…
99
100
```

```
1
2
3
4
hello world !
5
6
7
…
99
100
```

```
1
2
3
…
22
23
24
…
99
100
hello world !
```

Synchronization

to ensure synchronization, use :

- sleep()
- abort()
- join()
- interrupt()

locks & semaphores

several threads sharing a resource

critical section : exclusive execution

use a lock to ensure exclusivity

locks & semaphores

in each thread willing to access an exclusive resource :

```
object access_grant=new object();

lock(access_grant)
{
    critical section;
}
```

locks & semaphores

problem : *access_grant* is local to each thread : use a static object to share among threads

```
class myThread
{
     static object access_grant = new object();

     public void myMethod()
     {
         lock(access_grant)
         {
             critical section;
         }
     }
}
```

locks & semaphores

```
class test
{
    static void Main(string [] args)
    {
        myThread [] progs = new myThread[3];

        foreach (myThread m in progs)
        {
            new Thread(new
    ThreadStart(m.myMethod)).Start();
        }
    }
}
```

locks & semaphores

if a thread asks for access while already locked :

thread is queued wrt the access_grant object.

use the Monitor class to check access_grant status before entering critical section

Monitor class

```
object access_grant = new object();

Monitor.Enter(access_grant);
critical section;
Monitor.Exit(access_grant);
```

TryEnter method : if the resource is locked, do something else.

Monitor class

```
if (Monitor.TryEnter(access_grant))
// true if resource is available
{
    critical section code
}
else
{
    do something interesting anyway
}
```

Semaphore and Mutex

to synchronize threads and processus :use the Semaphore and Mutex classes

```
static Semaphore sem(0,n); // initial
   and max threads allowe to possess
   the semaphore
```

```
sem.WaitOne()
```

critical section

```
sem.Release()
```

Semaphore and Mutex

- a Mutex is a Semaphore with initial count=0 ans a max count = 1
- for more informations on semaphores, locks, monitors, see your OS documentation.

Working with processes

- a little break in theory
- how to list processes on the local computer
- use the System.Diagnostics classes and the Process class

Working with processes

```
class test
{
   static void Main(string [] args)
   {
      Process [] locals = Process.GetProcesses();

      foreach(Process p in locals)
      {
         System.Console.Write(p.ProcessName);
      }

      // pause
   }

}
```

Working with processes

```
class test
{
   static void Main(string [] args)
   {
      Process appex = new Process();

      appex.StartInfo.FileName = "path_to_exe";

      appex.StartInfo.UseShellExecute = false;

      appex.StartInfo.RedirectStandardOutput = false;

      appex.Start(); // target application starts here

      // pause
   }
}
```

Executing a standard application (any .exe) from a C# program

Networking made easy

use System.Net and System.Net.Sockets

C# : TcpListener and TcpClient classes

server

client

1 listen to connections
trough a TcpListener

2 connection to server
through a TcpClient

3 connection is accepted :
dialog through a TcpClient

4 dialog through TcpClients

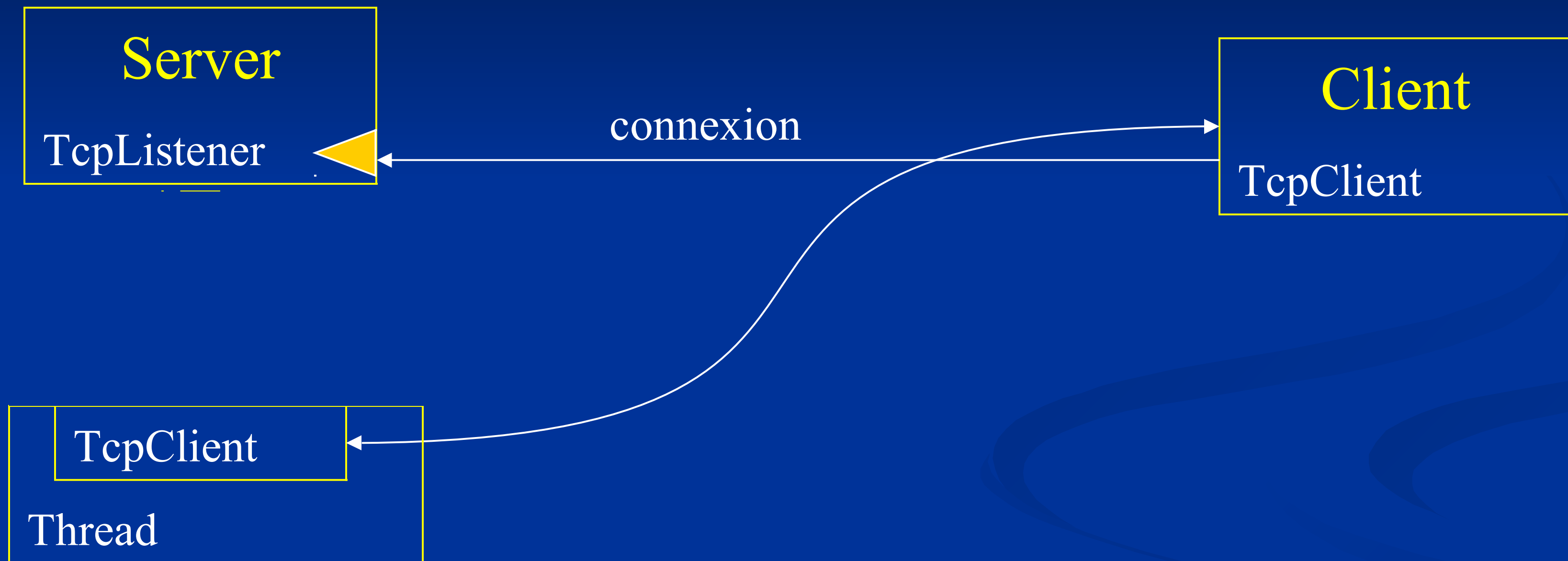
Sample server code

```
class server
{
  private TcpListener ear;
  private TcpClient cli_sock;

  public server()
  {
     byte [] local_IP ={127,0,0,1};

     ear = new TcpListener(new IPAdress(local_IP),8080);

     ear.start();

     cli_sock = ear.AcceptTcpClient(); // blocking

     // now talk with the client
  }
}
```

Sample client code

```
class client
{

  private TcpClient sock;

  public client()
  {

     sock = new TcpClient("server_name",8080);

     // now talk with the server if connected
  }
}
```

Multi client server



working with streams

input and output streams to read/write with TcpClient objects

```
StreamReader in=new StreamReader(sock.GetStream());

StreamWriter out=new StreamWriter(sock.getStream());

out.AutoFlush=true;
```

read data with `in.ReadLine();`

write data with `out.WriteLine();`

networks and threads

sockets (TcpClient objects) may send and receive information at any time :

use a thread to run a method that receives information :

```
while (connection is valid)
{
  read data sent by the server-side socket;
}
```

Communication

network applications :

using sockets and a communication protocol

- dedicated to the application;
- existing protocol : SMTP, FTP, HTTP, SOAP : Web development.