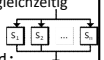
Nebenläufigkeit: con

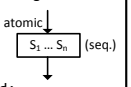
Programmteile werden nebeneinander, gleichzeitig ausgeführt




```
x = 0; y = 0;
con {x = x + 1; y = y + 1;} end;
z = x + y;
```

Atomare Teile: atomic

Ein Programmteil ist atomar wenn kein Zwischenschritt, kein Zwischenergebnis und kein Zwischenzustand in anderen parallel laufenden Programmteile sichtbar sind.



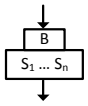

```
x = 0; y = 0;
con
  atomic {z = x + y;} end;
  atomic {x = 1; y = 2;} end;
end;
// wird atomar, parallel ausgeführt
-> Reihenfolge ist jedoch undefiniert -> Zufall!
```

Explizite Synchronisation: await

Explizite Synchronisation mit Bedingung, z.B. warten bis eine Initialisierung abgeschlossen ist.
Await ist per Definition atomic.

`await B [then S1 … Sn] end;`
`await s > 0 then s = s – 1; end;`

```
// in C# muss es auf Waiter und Tester aufteilen werden
Object synch = new Object();
// waiter
lock (synch) {
  Monitor.Wait(synch);
  S1; ...; Sn;
}
//tester
lock (synch) {
  //wenn Bedingung B wahr wird
  if (B) Monitor.Pulse(synch);
}
```



Korrektheit von Programmen:

Ein Programm ist korrekt, wenn es sicher (keine fehlerhaften Zustände eintreten) und lebendig (irgendwann alle gewünschten Zustände eintreten) ist.
- **Sicherheit**: kein Deadlock! Keine Interferenzen in kritischen Bereichen
- **Lebendigkeit**: kein Verhungern! Jeder Programmteil erhält eine faire Ausführung, kein Livelock (Starvation)
  -- **unconditionaly fair**: es gibt keine Bedingung
  -- **weak fair**: keine Garantie ob ausgeführt wird
  -- **strong fair**: garantierte Ausführung der Steps

unconditionaly fair

```
cont = true;
con
  while cont do
    /* nothing */
  end;
  cont = false;
end;
```

weak / strong fair

```
cont = true; try = false;
con
  while cont do
    try = true;
    //weak:   /* tick */
    //strong: Thread.Sleep(0);
    try = false;
  end;
  await try then cont = false; end;
end;
```

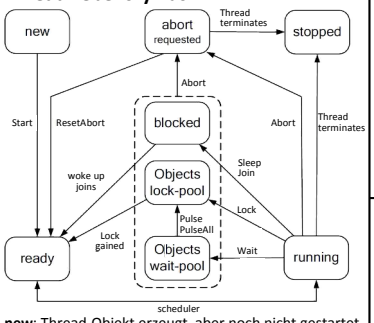
Producer / Consumer (Kopieren mit einem Puffer)

```
buf=0; p=0; c=0;
con
  producer;
  consumer;
end;
```

```
producer:
while p < n do
  await p = c end;
  buf = a[p+1];
  p++;
end;
```

```
consumer:
while c < n do
  await p > c end;
  b[c+1] = buf;
  c++;
end;
```

Thread Lebenszyklus:



new: Thread-Objekt erzeugt, aber noch nicht gestartet
ready: Thread ist gestartet, lokal Speicher (stack) ist zugeteilt, wartet auf die Zuweisung des Prozessors
running: führt Anweisungen auf dem Prozessors aus
blocked: Thread muss warten bis die Bedingung erfüllt ist, z.B. I/O, System-Call, anderer Thread
stopped: Thread existiert nicht mehr, Thread-Objekt jedoch schon und kann, falls es referenziert ist, benutzt werden. Andernfalls Garbage Collector: entfernt es
Wichtig: Der Objects lock-pool und der Objects wait-pool müssen zum gleichen Objekt gehören!
CLR: Common Language Runtime

Erzeugen von Threads:

Thread starten ohne Parameter

```
Thread t = new Thread(new ThreadStart(Go));
t.Start();
```

Thread starten mit Parameter

```
// Übergebener Parameter muss vom Typ Object sein!
static void Main(string[] args) {
  Thread t=new Thread(new ParameterizedThreadStart(Go));
  t.Start(true); // Thread mit Parameter «true» starten
}
static void Go(object upperCase) {
  bool upper = (bool)upperCase;
  Console.WriteLine(upper ? "HELLO!" : "hello!");
}
```

Thread starten mit Parameter (kurz Schreibweise)

```
Thread t = new Thread(Go);
t.Start(true);
```

Thread Priorität ändern

```
t.Priority = ThreadPriority.Lowest;
```

notenstatistik.ch

Beenden von Threads:

Thread starten ohne Parameter
Ein Thread ist beendet, wenn
- Thread-Methode ohne Fehler beendet
- Exceptions in Thread, z.B. Division durch Null
- Thread wird von aussen abgebrochen

t.Interrupt() `ThreadInterruptedException`
Interrupts werden von Threads nur bearbeitet, wenn er im blocked oder wait Zustand ist. Beim Aufruf im Running Mode läuft der Thread einfach weiter.
Exceptions müssen selber abgefangen werden.

```
try
  { Thread.Sleep(500); }
catch (ThreadInterruptedException)
  { Console.WriteLine("interrupt called..."); }
finally
  { … }
// überprüfen ob Thread im Wait/Sleep/Join Mode ist
if (t.ThreadState != ThreadState.Running)
```

t.Abort() `ThreadAbortException`
Es wird die Exception ThreadAbortException ausgelöst, die Methode wird ordnungsgemäss beendet.

```
try
  { Thread.Sleep(500); }
catch (ThreadAbortException)
  { Console.WriteLine("interrupt called..."); }
```

Thread.ResetAbort()
Bei t.Abort() wird ein Flag gesetzt, welches beim nächsten Running Zustand abgefragt wird und entsprechend die Exception ausgelöst. Mittels t.ResetAbort() kann der Thread wieder in den Ready Zustand versetzt werden.

```
try { while (true); }
catch (ThreadAbortException)
  { Thread.ResetAbort(); }
```

Einfache Synchronisation / Blockierung:

```
long sum = 0;  bool fertig = false;
Thread t = new Thread(delegate() {
  for (int i = 0; i <= 1000000000; i++)
    sum += i;
  fertig = true;
});
t.Start();
while (!fertig);  //v1:Threads switchen immer
Thread.Sleep(50); //v2:unsicher, evt. noch nicht fertig
t.Join();         //v3:beste Version
Console.WriteLine("Summe = " + sum);
```

Synchronisation / Lock-Konstrukt:

Threads reservieren einen Codebereich für sich. Beim betreten wird die Sperre gesetzt und beim verlassen wieder freigegeben. Falls sie bereits besetzt ist, wird gewartet bis sie frei wird. Lock() muss ein Object als Parameter besitzen (das gemeinsame Lock-Object).

```
static object locker = new object();
long sum = 0;
// Thread 1: berechnet die Summe
lock (locker) {
  sum = 42; }
// Thread 2: gibt die Summe aus
lock (locker) {
  Console.WriteLine("Summe = " + sum); }
```

WaitHandle (Manual/AutoResetEvent):

Mit wh.Set() wird der Status «aktiv», bei AutoResetEvent wird nur ein Thread geweckt und der Status zurück gesetzt. Beim ManualResetEvent bleibt der Status und alle Threads werden geweckt. Mittels wh.Reset() wird der Status zurück gesetzt.

```
class BasicWaitHandle {
  static EventWaitHandle wh =
                    new ManualAutorResetEvent(false);
  static void Main() {
    new Thread(Waiter).Start();
    Thread.Sleep(1000);
    wh.Set();
  }
  static void Waiter() {
    Console.WriteLine("Waiting...");
    wh.WaitOne();
    Console.WriteLine("Notified");
  }
}
// Systemweite Synchronisierung
EventWaitHandle signal = new EventWaitHandle(false,
EventResetMode.ManualReset, "hslu.prgsy.latch");
```

Wait / Pulse / PulseAll:

Die Threads warten an einem Monitorobjekt, bis dieses einen Impuls erhält, um einen oder mehrere Threads frei zu schalten. Bei einem Wait-Aufruf wird der Thread in den Wait-Pool geschickt und der Lock freigegeben. Wait und Lock-Pool müssen vom gleichen Objekt gestellt werden! Drei Wege aus dem Wait:
- anderer Thread signalisiert mittels Pulse / PulseAll
- TimeOut abgelaufen
- anderer Thread ruft die Methode Abort auf

```
static object synch = new Object(); // Lock-Objekt
// Waiter-Thread
lock (synch) { // Wait mit TimeOut
  if (Monitor.Wait(synch,1000))
    Console.WriteLine("Notified");
  else
    Console.WriteLine("Timeout");
}
// Main-Thread
lock (synch) { // Wartenden Thread freigeben
  Monitor.Pulse(synch); }
```

Wichtig: Wait und Pulse dürfen nur innerhalb von einem Lock-Block aufgerufen werden!
Das Signal wird nicht gespeichert!

Semaphore:

Allgemeines Konzept für die Synchronisation. Mutex nur Ja/Nein, Semaphore definieren die anzahl Threads die Zugriff auf einen kritischen Abschnitt haben sollen. Zwei wichtige Operationen: sema.P() und sema.V():
P(): «Proberen/Passieren» `await s>0 then s=s-1;end;`
V(): «Verhogen/Freigeben» `atomic s=s+1; end;`

```
sema.WaitOne(); // P(), kritischen Bereich betreten
sema.Release(); // V(), kritischen Bereich verlassen
// Semaphore(init, capacity); wie ein Lock: (1,1)
Semaphore sema = new Semaphore(1, 3); //init: 1, max: 3
while (true) {
  Console.WriteLine("Thread waits.");
  sema.WaitOne();
  Console.WriteLine("Thread is in critical section");
  Thread.Sleep(1000);//Only 3 threads here at once
  sema.Release();
  Console.WriteLine("Thread leaves.");
}
// Bestehende Semaphore verwenden / resp. neue erzeugen
try {
  sema = Semaphore.OpenExisting("ximit.ch-Semaphore");
} catch (WaitHandleCannotBeOpenedException) {
  sema = new Semaphore(0, 2, "ximit.ch-Semaphore");
}
```

Mutex / Wechselseitiger Ausschluss:

Nebenläufige Prozesse, Threads können nicht gleichzeitig auf Daten zugreifen.
mut.P(): `await m>0 then m=0; end;`
mut.V(): `await m<1 then m=1; end;`
Mutex ist Prozess übergreifend (Semaphoren nicht)!

```
mutex.WaitOne([timeout]) // P(), auf Mutex warten
mutex.ReleaseMutex()     // V(), Mutex freigeben

static Mutex mutex = new Mutex(false, "ximit.ch-Demo");
static void Main()
{
  //Wait 5 seconds, if another instance is running: exit
  if (!mutex.WaitOne(TimeSpan.FromSeconds(5), false)) {
    Console.WriteLine("Another instance is running.");
    return; //exit
  }
  try {
    Console.WriteLine("Running - press Enter to exit");
    Console.ReadLine();
  }
  finally { mutex.ReleaseMutex(); }
}
```

Bounded-Buffer (Anwendung):

n Produzenten erzeugen Daten, die in einem Puffer zwischengespeichert werden. m Konsumenten holen die Daten aus dem Puffer.
Synchronisation muss sicherstellen, dass Produzenten warten falls Puffer voll und Konsumenten warten falls Puffer leer ist.

```
x = 0; k = bufferSize;
con i = 1 to n do Producer(i);
con j = 1 to m do Consumer(j);

Producer(i):
while (true) {
  /* Daten erzeugen */
  await x < k then
  /* Daten in Puffer schreiben */
  x = x + 1;
end;
}

Consumer(j):
while (true) {
  await x >= 1 then
  /* Daten aus Puffer lesen */
  x = x - 1;
  end;
  /* Daten verbrauchen */
}
// Lesen & Schreiben auf x nicht gleichzeitig möglich!
// → Verwendung von zwei Zählern
empty = bufferSize; full = 0;
con i = 1 to n do Producer(i);
con j = 1 to m do Consumer(j);

Producer(i):
while (true) {
  /* Daten erzeugen */
  await empty > 0 then empty--; end;
  /* Daten in Puffer schreiben */
  atomic full++; end;
}

Consumer(j):
while (true) {
  await full > 0 then full--; end;
  /* Daten aus Puffer lesen */
  atomic empty++; end;
  /* Daten verbrauchen */
}
// Implementierung mit RingBufferArray
class RingBufferArray {
  protected Object[] array;
  protected int front = 0;
  protected int rear = 0;
  private Object putLock = new Object();
  private Object getLock = new Object();

  public RingBufferArray(int n) {
    array = new Object[n];
  }
  public void Put(Object x) {
    lock (putLock) {
      array[rear] = x;
      rear = (rear + 1) % array.GetLength(0);
    }
  }
  public Object Get() {
    lock (getLock) {
      Object x = array[front];
      array[front] = null;
      front = (front + 1) % array.GetLength(0);
      return x;
    }
  }
}
class BoundedBufferWithSemaphor : Iqueue {
  protected Semaphore empty;
  protected Semaphore full;
  protected RingBufferArray buf;

  public BoundedBufferWithSemaphor(int size) {
    buf = new RingBufferArray(size);
    empty = new Semaphore(size,size);
    full = new Semaphore(0,size); }
  public void Enqueue(Object x) {
    empty.WaitOne();
    buf.Put(x);
    full.Release(); }
  public Object Dequeue() {
    full.WaitOne();
    Object x = buf.Get();
    empty.Release();
    return x;
  }
}
```

Parkhaus:

```
class Parkhaus {
  private int n;
  public Parkhaus(int p) {
    n = p; }
  public void Passieren() {
    lock (this) {
      while (n == 0) {
        Monitor.Wait(this); }
      n--; }}
  public void Verlassen() {
    lock (this) {
      n++;
      Monitor.PulseAll(this); }}
  public int AnzahlPlaetze() {
    lock (this)
      return n; }
  public static void Main() {
    Parkhaus parkhaus = new Parkhaus(10);
    for (int i = 1; i <= 60; i++) {
      Auto auto = new Auto("Auto " + i, parkhaus);
      new Thread(auto.Drive).Start();
}}}

class Auto
{ […]
  public void Drive() {
    while (true) {
      Thread.Sleep(rnd.Next(3000));
      lock (parkhaus) {
        parkhaus.Passieren();
        Console.WriteLine(name + ": eingefahren \t" +
                          parkhaus.AnzahlPlaetze());
      }
      Thread.Sleep(rnd.Next(1000));
      lock (parkhaus) {
        parkhaus.Verlassen();
        Console.WriteLine(name + ": ausgefahren");
      }
}}}
```

Asynchrone Aufrufe mit Callback:

Asynchrone Aufrufe müssen in C# durch Nebenläufigkeit hergestellt werden.
+ weniger Traffic, Zeitnahe Info
- keine Kontrolle über Handler

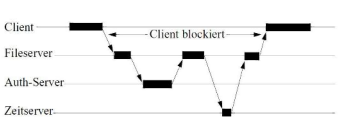
```
interface ICallbackHandler {
  void Handle(Object res);
}
class Service : ICallbackHandler {
  public void Request() {
    // Die Aufgabe und eine Referenz auf sich selbst
    // an den Handler übergeben und starten...
    Handler handler = new Handler(this);
    new Thread(handler.Do).Start();
    // weitere Aufgaben bearbeiten
  }
  public void Handle(Object res) {
    // Resultat steht zur Verfügung
    // oder als Parameter erhalten.
  }
}
class Handler {
  ICallbackHandler service;
  public Handler(ICallbackHandler service) {
      this.service = service; }
  public void Do() {
    // Hier ist die Aufgabe zu lösen...
    service.Handle(result);
  }
}
```

Asynchrone Aufrufe mit Polling:

```
// Die Aufgabe dem Handler übergeben und starten...
Thread worker = new Thread(handler.Do);
worker.Start();
// ...auf das Resultat warten...
while(worker.IsAlive) {
  // Hier können andere Aufgaben bearbeitet werden...
}
// ...Resultat steht zur Verfügung.
handler.GetResult();
```

Executor / Worker Threads:

```
//Mandelbrot
private IExecutor exec;
exec = new PlainWorkerPool(
                new BoundedBufferWithSemaphor(60000),
                Environment.ProcessorCount);

MandelbrotMenge mm = new MandelbrotMenge(this, a, b);

//new Thread(mm.Calc).Start();
//mm.Calc();
exec.Execute(mm.Calc);

//-----------------------------------------------------
//Worker Threads mit RingBufferArray
internal interface Iqueue {
  void Enqueue(Object x);
  Object Dequeue();
}

internal interface Iexecutor {
  void Execute(ThreadStart threadStart);
}

internal interface ICallbackHandler {
  void Handle(Object res);
}

internal class TestWorkerPoolExec {
  public static void Main() {
    PlainWorkerPool executor = new PlainWorkerPool(
          new BoundedBufferWithSemaphor(100), // Queue
          10); // Anzahl Workers
    Service service = new Service(executor);
    for (int i = 0; i < 10; i++) {
      service.Request();
    }
  }
}

internal class PlainWorkerPool : Iexecutor {
  protected IQueue workQueue;

  public PlainWorkerPool(IQueue workQueue,
                         Int nWorkers) {
    this.workQueue = workQueue;
    for (int i = 0; i < nWorkers; ++i) {
      activate();
    }
  }

  public void Execute(ThreadStart threadStart) {
    workQueue.Enqueue(threadStart);
  }

  protected void activate() {
    Thread runLoop = new Thread(delegate(){
      while (true) {
        ThreadStart threadStart =
                    (ThreadStart) workQueue.Dequeue();
        threadStart.Invoke();
      }
    });
    runLoop.Start();
  }
}

class Service : ICallbackHandler {
  protected IExecutor executor;

  public Service(IExecutor executor) {
    this.executor = executor;
  }

  public void Request() {
    //Die Aufgabe und Referenz auf sich selbst
    //an den Handler übergeben und starten...
    Handler handler = new Handler(this);
    executor.Execute(handler.Do);
    for (int i = 0; i < 5; i++) {
      Console.WriteLine("weitere Aufgaben abarbeite");
      Thread.Sleep(300);
    }
  }

  public void Handle(Object res) {
    Console.WriteLine("Resultat zur Verfügung: " +
                      res.ToString());
  }
}

internal class Handler {
  private ICallbackHandler service;

  public Handler(ICallbackHandler service) {
    this.service = service;
  }

  public void Do() {
    Console.WriteLine("Die Aufgabe lösen...");
    Thread.Sleep(1000);
    Console.WriteLine("Aufgabe beendet,
                       Service benachrichtigen");
    service.Handle("Fertig.");
  }
}
```



Streams:

Base-Streams: lesen und schreiben
Pass-Through-Streams: ergänzen die Base-Streams, eigene Zusammenstellungen sind möglich
Zeichen/Textorientiert: Stream-/String- Reader/Writer
Binär: BinaryReader / BinaryWriter
Elementare Operationen von Streams:
- Dateninformationen in einen Stream schreiben
- Aus dem Datenstrom lesen
- Wahlfreien Zugriff beim lesen (nicht immer von A-Z)
Wichtig: stream.flush() nach dem schreiben!

Stream Beispiele:

```
using System.IO;
using System.Net.Sockets;
```

Lesen / Schreiben auf Console

```
string line;
Console.Write("Bitte Eingabe: ");
while ((line = Console.ReadLine()) != null) {
  Console.WriteLine("Eingabe war : " + line);
  Console.Write("Bitte Eingabe: "); }
```

Lesen / Schreiben in ein TCP-Socket

```
TcpClient client = new TcpClient("192.53.103.103", 13);
StreamReader inStream =
                    new StreamReader(client.GetStream());
StreamWriter outStream =
                    new StreamWriter(client.GetStream());
outStream.WriteLine("Hello from Server!");
outStream.Flush();
Console.WriteLine(inStream.ReadLine());
client.Close();
```

Schreiben in eine Datei mit FileStream

```
try {
  FileStream fs =
          new FileStream("daten.txt", FileMode.Create);
  StreamWriter sw = new StreamWriter(fs);
  string[] text = { "Titel", "Köln", "4711" };
  for (int i = 0; i < text.Length; i++) {
    sw.WriteLine(text[i]); }
  sw.Close();
  Console.WriteLine("fertig.");
} catch (Exception e) { Console.WriteLine(e); }
```

Schreiben in eine Datei mit implizitem FileStream

```
try {
  using (StreamWriter sw= new StreamWriter("daten.txt"))
  {
    string[] text = { "Titel", "Köln", "4711" };
    for (int i = 0; i < text.Length; i++)
      sw.WriteLine(text[i]);
  }
  Console.WriteLine("fertig.");
} catch (Exception e) { Console.WriteLine(e); }
```

Lesen aus einer Datei

```
try {
  using (StreamReader sr= new StreamReader("daten.txt"))
  {
    string line;
    while ((line = sr.ReadLine()) != null) {
      Console.WriteLine(line); }
  }
} catch (Exception e) { Console.WriteLine(e); }
```

Lesen aus einer Datei mit einem Pass-Through-Stream

```
using System.Security.Cryptography;

Stream stm = new FileStream("daten.txt", FileMode.Open,
                            FileAccess.Read);
ICryptoTransform ict = new ToBase64Transform();
CryptoStream cs =
      new CryptoStream(stm, ict, CryptoStreamMode.Read);
TextReader tr = new StreamReader(cs);
string s = tr.ReadToEnd();
Console.WriteLine(s);
```

Serialisierung: [Serializable]

BinaryFormatter / SoapFormatter / XmlSerializer
- vollständig qualifizierte Klassennamen des Obj.
- Signatur der Klasse
- Alle nicht statischen, nicht transienten Attribute des Objekts inkl. Aller aus den Oberklassen geerbten Attribute sowie die Attribute aller assoziierten Obj.

BinaryFormatter

```
using System.Runtime.Serialization.Formatters.Binary;

public void SerializeObject(String filename,Object obj){
  FileStream fs=newFileStream(filename,FileMode.Create);
  BinaryFormatter binFormatter = new BinaryFormatter();
  binFormatter.Serialize(fs, obj);
  fs.Close(); }
public Object DeserializeObject(String filename) {
  FileStream fs=new FileStream(filename, FileMode.Open);
  BinaryFormatter binFormatter = new BinaryFormatter();
  return binFormatter.Deserialize(fs); }
```

XmlSerializer
- Die zu serialisierende Klasse muss public sein
- Klasse: public, Parameterlosen Constructor
- Es werden nur public Attribute serialisiert

```
using System.Xml.Serialization;

public void SerializeObj(String fileame, MyClass obj) {
 FileStream fs=new FileStream(filename,FileMode.Create);
 XmlSerializer xml = new XmlSerializer(typeof(MyClass));
 xml.Serialize(fs, obj);
 fs.Close(); }
public Object DeserializeObject(String filename) {
 FileStream fs=new FileStream(filename, FileMode.Open);
 XmlSerializer xml = new XmlSerializer(typeof(MyClass));
 return xml.Deserialize(fs); }
```

DeepCopy

```
public static Object DeepCopy(Object obj) {
  MemoryStream ms = new MemoryStream(500);
  BinaryFormatter binFormatter = new BinaryFormatter();
  binFormatter.Serialize(ms, obj);
  ms.Position = 0; //ms.Seek(0, 0);
  return binFormatter.Deserialize(ms); }
```

Socket

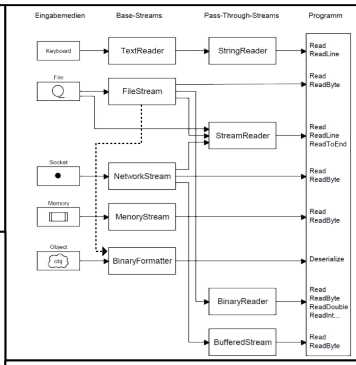
- Verbindung zu entferntem Prozess aufbauen
- Daten senden / Daten empfangen
- Verbindung schliessen
- einen Port (Prozess) binden
- an einem Port auf Verbindungswunsch hören
- Verbindungswunsch an Port akzeptieren

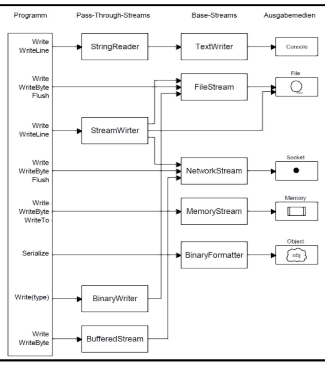
ClientSocket

```
using System.Net.Sockets;
try {
  TcpClient client = new TcpClient("10.0.0.1", 80);
  Socket clientSocket = client.Client;
  Console.WriteLine("Connected to: " +
                    clientSocket.RemoteEndPoint);
  client.Close();
} catch (Exception e) { Console.WriteLine(e); }
```

ServerSocket

```
//TcpListener listen = new TcpListener(13);
TcpListener listen = new TcpListener(IPAddress.Any, 13);
listen.Start();
while (true) {
  Console.WriteLine("Warte auf Verbindung auf Port " +
                    listen.LocalEndpoint + "...");
  TcpClient client = listen.AcceptTcpClient();
  Console.WriteLine("Verbindung zu " +
                    client.Client.RemoteEndPoint);
  StreamWriter sw =new StreamWriter(client.GetStream());
  sw.Write(DateTime.Now.ToString());
  sw.Flush();
  client.Close(); }
```





Server Typen

Iterativer Server
- 1. Warte auf Request von einem Client
- 2. Auftrag ausführen
- 3. Antwort an Client zurücksenden
- 4. Gehe zu Schritt 1

Nebenläufiger Server
- Multi-Thread Server
- Master (Listener) wartet auf Request von Client
- Master erzeugt für jeden Client ein Slave-Thread
- Slave (Handler) führt den Auftrag aus
- Slave schickt die Antwort zurück an den Client
- ggf. Slave reuse -> ThreadPool, sonst terminieren

AbstractServer

Die Run Methode...
- erzeugt zur Skalierung Executor (CreateExecutor)
- erzeugt Server Socket Objekt (CreateServerSocket)
- Client-Anfragen mit Cli-Socket entgegen (Accept)
- erzeugt Handler-Objekt (CreateHandler)
- startet Handler-Objekt-Thread (Execute)

ConcreteServer

Ein konkreter Server kann, bzw. muss die folgenden Methoden implementieren:

- **`CreateExecutor`** – Methode zur Erzeugung eines Executors. Mit dem Executor hat man die Möglichkeit den Server skalierbar zu machen. Je nach Bedürfnis kann der Executor mit verschiedenen Buffern und unterschiedlicher Anzahl von ausführenden Threads initialisiert werden. Mit der Standardimplementierung wird ein einfacher Thread-Executor ohne Buffer erzeugt.

- **`CreateServerSocket`** – Methode zur Erzeugung eines Serversockets. Diese Methode eröffnet die Möglichkeit einen Server Socket auf eine spezifische unterschiedliche Art zu erzeugen. Mit der Standardimplementierung wird ein TcpListener-Objekt am definierten Port erzeugt.

- **`CreateHandler`** – Methode zur Erzeugung eines Handlers. Der Handler bestimmt die Funktion des Servers und muss daher mit durch eine konkrete Handler-Klasse implementiert werden.

```
namespace Executor {
  public interface Iexecutor {
    void Execute(ThreadStart threadStart);
  }
  public class PlainThreadExecutor : Iexecutor {
    public void Execute(ThreadStart threadStart) {
      new Thread(threadStart).Start(); }
  }
  public class WorkerPool : Iexecutor {
    public WorkerPool(int nWorkers) {
      ThreadPool.SetMaxThreads(nWorkers,nWorkers);
    }
    public void Execute(ThreadStart threadStart) {
      ThreadPool.QueueUserWorkItem(
                  new WaitCallback(Work),threadStart);
    }
    protected void Work(Object todo) {
      ThreadStart threadStart = (ThreadStart)todo;
      threadStart.Invoke();
    }
}
namespace ServerPattern {
  public abstract class AbstractServer {
    private Boolean running = true;
    private int port;
    private string host;
    public const int DEFAULTPORT = 4711;
    public const string DEFAULTHOST = "localhost";

    protected void Run() {
      IExecutor executor = CreateExecutor();
      Socket listen = CreateServerSocket();
      while (running) {
        AbstractHandler handler =
                                CreateHandler(listen.Accept());
        executor.Execute(handler.Run);
    }}
    virtual protected IExecutor CreateExecutor() {
      return new PlainThreadExecutor();
    }
    virtual protected Socket CreateServerSocket() {
      IPAddress ipAddress =
                    Dns.GetHostEntry(host).AddressList[0];
      TcpListener listener =
                          new TcpListener(ipAddress, port);
      listener.Start();
      return listener.Server;
    }
    public void Start() {
      this.Start(DEFAULTPORT,DEFAULTHOST);
    }
    public void Start(int port) {
      this.Start(port,DEFAULTHOST);
    }
    public void Start(int port, string host) {
      if ((port <= 0) || (host == null)) throw new
                  ArgumentOutOfRangeException("host/port");
      Thread server = new Thread(Run);
      this.port = port;
      this.host = host;
      server.Start();
    }
    public void Stop() {
      running = false;
      new TcpClient(host, port);
    }
    public int GetPort() { return port; }
    public string GetHost() { return host; }

    abstract protected AbstractHandler
                                CreateHandler(Socket client);
  }
}
```

AbstractHandler

- verbindet Client-Socket mit Input- / Output-Stream
- liest Request aus Input-Stream und wertet dieser aus
- generiert entsprechende Antwort und sendet diese via Output-Stream dem Client zurück

ConcreteHandler

Ein konkreter Handler muss einen Konstruktor und die folgenden zwei Methoden implementieren:

- **`Konstruktor`** – Er verbindet den Client-Socket mit einem konkreten Input- und Outputstream. Das heisst, der konkrete Handler muss konkrete Input- und Outputstreams (z.B. StreamReader, StreamWriter oder BinaryWriter, BinaryReader, etc.) als Attribute definieren, sodass die Methoden ReadRequest und CreateResponse darauf zugreifen können.

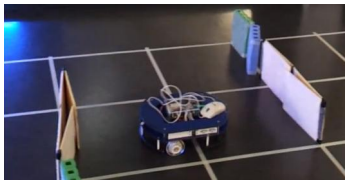
- **`ReadRequest`** – Methode zum Lesen der Anfrage des Klienten. Sie liest aus dem konkreten Inputstream Daten oder Anweisungen aus, welche für die Antwort benötigt werden.

- **`CreateResponse`** – Methode zur Erzeugung einer entsprechenden Antwort. Über den konkreten Outputstream werden die Daten (oder Objekte) der Antwort an den Klienten gesendet.

```
namespace ServerPattern {
  public abstract class AbstractHandler {
    private Socket client;

    public AbstractHandler(Socket client) {
      this.client = client;
    }
    public void Run() {
      try {
        if (ReadRequest()) { CreateResponse(); }
      } catch (Exception e) { Console.WriteLine(e);
      } finally { client.Close(); }
    }

    abstract protected Boolean ReadRequest();
    abstract protected void CreateResponse();
}}
```

HelloServer (ConcreteServer/Handler)

```
namespace Hello {
  class HelloServer : AbstractServer {
    override protected AbstractHandler
                                CreateHandler(Socket client) {
      return new HelloHandler(client);
    }
    public static void Main() {
      new HelloServer().Start();
      Console.WriteLine("Server auf {0}gestartet.",
                ServerPattern.AbstractServer.DEFAULTPORT);
    }
  }

  class HelloHandler : AbstractHandler {
    private StreamReader sr;
    private StreamWriter sw;

    public HelloHandler(Socket client) : base (client) {
      NetworkStream nws =
                                new NetworkStream(client,true);
      sr = new StreamReader(nws);
      sw = new StreamWriter(nws);
    }
    override protected Boolean ReadRequest() {
      String request = sr.ReadLine();
      return true;
    }
    override protected void CreateResponse() {
      sw.WriteLine("Hello world.");
      sw.Flush();
    }
  }
}
```



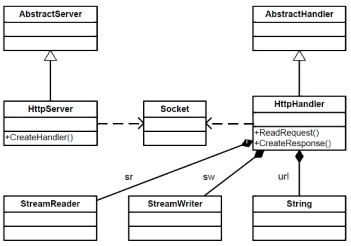
Client für DeepCopy-Server

```
using System.Runtime.Serialization.Formatters.Binary;
using System.Net.Sockets;

public static void Main() {
  TcpClient client = new TcpClient("localhost", 7788);
  BinaryFormatter binFormatter = new BinaryFormatter();
  NetworkStream stream =
                          new NetworkStream(client.Client);
  binFormatter.Serialize(stream,
                          new MyClass("Luzern", 6006));
  stream.Flush();
  MyClass t = (MyClass)binFormatter.Deserialize(stream);

  Console.WriteLine("ID: " + t.getId() +
                    ", Value: " + t.getValue());
}
```

EventWaitHandle Array

Ein Array von EventWaitHandles. Jedes ClientObject erhält ein Array-Element und ruft in der Methode Latch.Acquire() das «set» des EventWaitHandles auf. ServerObject wartet mit WaitAll auf alle Array-Elemente.

Klassenstruktur HTTP-Server



Implementation HTTP-Server (Server)

```
class HttpServer : AbstractServer {
  private int call = 0;

  override protected AbstractHandler
                                CreateHandler(Socket client) {
    return new HttpHandler(client, ++call);
  }

  override protected IExecutor CreateExecutor() {
    return new WorkerPool(20);
  }

  public static void Main(String[] args) {
    int port = 8080;
    if (args.Length > 0) {
      port = Int32.Parse(args[0]);
    }
    new HttpServer().Start(port);
    Console.WriteLine("Server run on port: " + port);
  }
}
```

Implementation HTTP-Server (Handler)

```
class HttpHandler : AbstractHandler
{
// Multipurpose Internet Mail Extensions (MIME)
// Internet Erweiterungen für die Einbindung von Daten.
private string[,] mimetypes = {
  {"html","text/html"},
  {"htm", "text/html"},
  {"txt", "text/plain"},
  {"gif", "image/gif"},
  {"jpg", "image/jpeg"},
  {"jpeg","image/jpeg"}
};

private int id;
private NetworkStream nws;
private StreamReader sr;
private StreamWriter sw;
private string url;

public HttpHandler(Socket client, int id):base(client){
  this.id = id;
  nws = new NetworkStream(client, true);
  sr = new StreamReader(nws);
  sw = new StreamWriter(nws);
}

override protected bool ReadRequest() {
  Console.WriteLine(id+". Incoming request...");
  string headerline;
  ArrayList request = new ArrayList();
  // Request-Header-Zeilen lesen bis zur Leerzeile
  while ((headerline = sr.ReadLine()) != null
          && headerline != "") {
    request.Add(headerline);
    Console.WriteLine("< " + headerline);
  }
  // 1. Request-Zeile auf HTTP Methode GET untersuchen
  string[] tokens =
        ((string)request[0]).Split(new char[] { ' ' });
  if (tokens.Length >= 2 && tokens[0] == "GET") {
    // URL ermittlen
    if (tokens[1].StartsWith("/"))
      url = tokens[1];
    // Start URL setzen
    if (url.EndsWith("/"))
      url += "index.html";
    return true;
  } else {
    WriteError(400, "Bad Request");
    return false;
  }
}

override protected void CreateResponse() {
  try {
    string filename = url.Substring(1);
    FileStream fs = new FileStream(filename,
                                   FileMode.Open, FileAccess.Read);
    long len = fs.Length;
    byte[] bytes = new byte[len];
    fs.Read(bytes, 0, (int)len);
    // Alles OK
    WriteResult("HTTP/1.0 200 OK");
    // Servererkennung senden
    WriteResult("Server: XimitWebServer 1.0");
    // Content-Length ermitteln und senden
    WriteResult("Content-Length: " +
                bytes.Length.ToString());
    // Content-Type ermitteln und senden
    // Beginnen mit unbekanntem Dateityp
    String mimestring = "application/octet-stream";
    for (int i = 0; i < mimetypes.Length/2; ++i) {
      if (url.EndsWith(mimetypes[i,0])) {
        mimestring = mimetypes[i,1];
        break;
      }
    }
    WriteResult("Content-Type: " + mimestring);
    // Leerzeile senden
    WriteResult("");
    // Daten senden
    nws.Write(bytes, 0, (int)len);
    nws.Flush();
    fs.Close();
  }
   catch (FileNotFoundException) {
     WriteError(404, "File Not Found"); }
   catch (DirectoryNotFoundException) {
     WriteError(404, "File Not Found"); }
   catch (Exception e) {
     WriteError(410, "Unknown Exception");
     Console.Error.WriteLine(e);
   }
}

public void WriteResult(string message) {
  Console.WriteLine("> " + message);
  sw.Write(message+"\r\n");
  sw.Flush();
}

public void WriteError(int status, string message) {
  string output = "<h1>HTTP/1.0 " + status + " " +
                  message + "</h1>";
  Console.WriteLine("> " + status.ToString() + " " +
                    message);
  sw.Write(output);
  sw.Flush();
} }
```